REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE
UNIVERSITE D'ORAN1 AHMED BENBELLA
FACULTE DES SCIENCES EXACTES ET APPLIQUEES
DEPARTEMENT D'INFORMATIQUE

THÈSE

Présentée par

Salim CHEHIDA

Pour obtenir le diplôme de

DOCTORAT EN SCIENCES

Spécialité

INFORMATIQUE

Thème

Approche de spécification et validation de politiques RBAC au niveau des processus métiers

Soutenue publiquement le 02/03/2017 devant le jury composé de :

Mr *Hafid HAFFAF*, Professeur à l'Université d'Oran1 Ahmed BenBella (Président du jury)

Mr *Mustapha Kamel RAHMOUNI*, Professeur à l'Université d'Oran1 Ahmed BenBella (Encadreur)

Mr *Allaoua CHAOUI*, Professeur à l'Université Abdelhamid Mehri Constantine 2 (Examinateur)

Mr *Djamel Eddine SAIDOUNI*, Professeur à l'Université Abdelhamid Mehri Constantine 2 (Examinateur)

Mr *Yahia LEBBAH*, Professeur à l'Université d'Oran1 Ahmed BenBella (Examinateur)

Mr Adda ALI PACHA, Professeur à l'Université USTO-MB (Examinateur)

Mr *Yves LEDRU*, Professeur à l'Université Grenoble Alpes (Invité)

Mr *Akram IDANI*, Maître de conférences à Grenoble INP (ENSIMAG) (Invité)

Avant-propos

Cette thèse de doctorat a été réalisée dans le cadre d'une collaboration entre l'université d'*Oran1 Ahmed BenBella*- Algérie et l'université *Joseph Fourier* –France, avec le soutien financier du projet P.N.E (Programme National Exceptionnel) du Ministère Algérien de l'Enseignement Supérieur et de la Recherche Scientifique. Les travaux de recherche ont été réalisés, en France, au sein du Laboratoire d'Informatique de Grenoble (l'équipe VASCO), et en Algérie, au sein du Département d'Informatique de l'Université d'Oran 1.

Remerciements

Je remercie vivement mon directeur de thèse, Mr *Mustapha Kamel RAHMOUNI* d'avoir accepté de diriger ce projet et de m'avoir accompagné et encouragé, tout en me donner l'opportunité de trouver par moi-même mon propre chemin.

Je tiens également à remercie très chaleureusement mes encadreurs en France :

- Mr *Yves Ledru*, Professeur à l'Université Grenoble Alpes et responsable de l'équipe VASCO du Laboratoire d'Informatique de Grenoble.
- Mr *Akram Idani*, Maître de conférences à Grenoble INP (ENSIMAG) et membre du Laboratoire d'Informatique de Grenoble (Equipe VASCO).

Je les remercie de m'avoir accueillie dans l'équipe VASCO, pour leur confiance et leur soutien, pour le temps et la patience qu'ils m'ont accordé et pour leurs conseils précieux et leurs commentaires judicieux qui ont largement contribué aux avancements de mon projet. Qu'ils trouvent ici l'expression de ma profonde gratitude pour leur personne et leur gentillesse, et ma haute considération pour leur compétence.

J'exprime aussi tous mes sincères remerciements aux membres du jury qui m'ont fait l'honneur de juger ce travail. Mr *Hafid HAFFAF*, Professeur à l'Université d'Oran1 qui m'a fait l'honneur de présider le jury de ma thèse. Mr *Allaoua CHAOUI*, Professeur à l'Université de Constantine 2, Mr *Djamel Eddine SAIDOUNI*, Professeur à l'Université de Constantine 2, Mr *Yahia LEBBAH*, Professeur à l'Université d'Oran1, et Mr *Adda ALI PACHA*, Professeur à USTO-MB d'avoir accepté la lourde charge d'être examinateurs de mon travail. Je les remercie pour les précieuses remarques et les critiques très constructives. Leurs suggestions m'étaient très intéressantes. Soyez assurés, messieurs, de ma profonde gratitude.

Un grand merci à mes amis et mes collègues *Flitti Touati, Benameur Abdelkader, Bensalloua Charef, Besnassi Miloud, Bouadjemi Abdelkarim, Khelifa Noreddine Belghiat Issam* et *Hettab AbdelKamel* pour leur disponibilité et leur soutien continu.

Merci particulièrement à *Deghdak Hocine* et sa femme *Sabrina, Houssem Chihoub* et *Abbas Bradai* qui m'ont soutenu durant les phases cruciales de mon stage à Grenoble loin de ma grande famille et loin du pays natal.

Enfin je remercie l'ensemble de mes professeurs qui m'ont inculqué leur savoir et leur expérience et tous ceux qui de près ou de loin ont contribué à la réalisation de ce travail.

Dédicace

Je dédie ce travail à toutes les personnes qui me sont très chères :

Mes parents : « vous représentez pour moi la clé de ma réussite, je vous suis reconnaissant de l'éducation que vous m'avez transmise, tes conseils et ton soutien tout au long de mes études »

Ma femme et mon petit bijou *Mohamed Anes*, qui m'ont soutenu et encouragé tout au long de la réalisation de ce travail. Je les remercie pour leur patience et leurs prières.

Mon cher frère et mes chères sœurs, mon oncle et mes tantes, mes cousins, toute ma famille et tous mes amis.

Résumé

La spécification de politiques de contrôle d'accès constitue une étape cruciale dans la sécurité d'un Système d'Information. Le présent travail propose une approche qui combine les langages UML et B pour la spécification et la validation de politiques de contrôle d'accès basées sur le modèle RBAC (Role Based Access Control) au niveau des activités d'un processus métier.

Notre approche commence par la spécification semi-formelle des règles de contrôle d'accès à l'aide de notre extension de diagrammes d'activités d'UML2 nommée BAAC@UML (Business Activity Access Control with UML). Les diagrammes d'activités sont définis à deux niveaux : un niveau abstrait qui ne détaille pas les règles d'autorisation et un niveau concret où des contraintes sont associées à certaines actions ou à l'ensemble du diagramme. Notre profil introduit les concepts de *rôle* et *précondition contextuelle de contrôle d'accès* pour garder l'accès des utilisateurs aux activités.

Nous avons défini un méta-modèle en vue de spécifier la sémantique de nos diagrammes d'activités BAAC@UML ainsi que la sémantique de leurs liens avec les modèles SecureUML qui permettent la spécification de la vue statique d'une politique RBAC. Nous avons également proposé un ensemble de règles, exprimées en OCL, qui permet d'assurer la cohérence structurelle entre les modèles BAAC@UML et les modèles SecureUML.

Les modèles BAAC@UML sont ensuite traduits en spécifications formelles exprimées en langage B. La spécification B définit un cadre rigoureux favorisant la vérification et la validation (V&V) de la politique RBAC. Nous avons utilisé l'animateur ProB et le prouveur Atelier B dans les activités de V&V. La preuve assure la conservation des contraintes de sécurité invariantes dans l'exécution de l'activité. L'animation permet de simuler l'exécution des scénarios d'activité par les utilisateurs et de vérifier que la politique RBAC est bien respectée au niveau des modèles d'activités.

MOTS-CLÉS : RBAC, Processus métier, SecureUML, Diagramme d'activités d'UML2, Cohérence, B, Animation, Preuve.

Abstract

The specification of an access control policy is a crucial activity in the security of an Information System. This work presents an approach that combines the UML and B languages for the specification and validation of RBAC (Role Based Access Control) policies at the business process level.

Our approach starts by modeling the access control rules using our extension of UML2 activity diagrams denominated as BAAC@UML (Business Activity Access Control with UML). Activity diagrams are defined at two levels: an abstract level which does not detail the authorization rules and a concrete level where constraints are associated to specific actions or to the whole diagram. Our profile introduces the *role* and *access control precondition* concepts to guard access of the user to the business process activities.

A meta-model has been defined in order to specify the semantics of our BAAC@UML diagrams and the semantics of their links with the SecureUML models that allow the specification of the static view of the RBAC policy. We have also proposed a set of rules, expressed in OCL, to ensure the structural consistency between the BAAC@UML models and the SecureUML models.

The BAAC@UML models are then translated into a specification in the B language. The B specification defines a rigorous framework for the verification and validation (V&V) of the RBAC policy. We have used the ProB animator and the Atelier B prover in the V&V activities. The proof ensures that the security invariants are preserved during the execution of the activity. The animation allows to simulate the execution of the activity scenarios by the users and to verify that the RBAC policy is respected in the activity models.

KEYWORDS: RBAC, Business process, SecureUML, UML2 Activity Diagram, Consistency, B, Animation, Proof.

Table des matières

Chapitre 3 : Cohérence entre les modèles BAAC@UML et SecureUML

Chapitre 4 : Spécification et validation formelles de modèles BAAC@UML

Chapitre 5 : Mise en œuvre des modèles BAAC@UML dans des outils

Conclusion générale

Liste des figures et tableaux

Introduction générale

La problématique de la sécurité des Systèmes d'Information (SI) est devenue une préoccupation majeure ces dernières années. L'ouverture de ces systèmes vers l'extérieur et l'augmentation massive de leurs utilisateurs rendent la protection de leurs informations de plus en plus complexe. La prise en compte des exigences de sécurité (telles que l'intégrité, la confidentialité, le contrôle d'accès, etc.), dès les premières étapes de cycle de développement, constitue l'un des principaux challenges pour les concepteurs des SI. Dans ce travail, nous proposons une approche qui décrit le déroulement des activités métiers d'un SI en tenant compte d'une politique de sécurité. Notre approche élabore la politique de sécurité en même temps que la modélisation fonctionnelle, et produit des modèles de processus métier qui intègrent à la fois les spécifications fonctionnelles et de sécurité.

I.1 Contexte et problématique

De nos jours, le succès d'une entreprise dépend de la qualité de son SI. Pour beaucoup d'entreprises, le SI est considéré comme une ressource importante qui constitue le maillon principal de sa chaîne de valeur ajoutée. Il correspond aux moyens mis en œuvre par l'entreprise pour traiter les informations. La description des processus métiers occupe une place centrale dans les SI d'entreprise. Un processus métier est défini comme étant une combinaison d'un ensemble d'activités mises en œuvre par des acteurs au sein d'une entreprise dans le but de produire un résultat déterminé. L'analyse et la conception des processus métiers permettent l'obtention d'une vue simplifiée facilitant la compréhension des SI et fournissant un excellent moyen pour leur formalisation et leur mise en œuvre. Elles donnent également aux entreprises la capacité d'adaptation et de maintenance de leurs systèmes dans un monde économique dynamique et concurrentiel.

Plusieurs langages graphiques ont été proposés pour la représentation des processus métiers. Actuellement, UML (UML2, 2011) et BPMN (*BPMN2*, 2011) sont les notations les plus répandues chez les industriels et les chercheurs. Cependant, si ces notations sont bien adaptées à la modélisation fonctionnelle des processus métiers, elles ne permettent pas d'exprimer les exigences de sécurité de ces processus. Avec la considérable augmentation des risques inhérents aux SI, la sécurité est devenue un aspect fondamental qui doit être pris en compte dans toutes les phases de développement. Face aux problèmes de sécurité, des nouvelles lois et normes très

strictes ont obligé les organisations et les entreprises à sécuriser les données présentées dans leurs SI. Ainsi des moyens considérables ont été mis en place.

Le contrôle d'accès est l'une des principales solutions face aux problèmes de sécurité. Il permet de spécifier des règles d'accès aux systèmes en définissant qui peut accéder à quelle ressource et sous quelles conditions. Plusieurs modèles ont été proposés pour l'expression des règles d'accès. Adopté comme une norme ANSI/INCITS (ANSI, 2004), le modèle RBAC (Role-Based Access Control) (Ferraiolo et al., 2003) est le modèle le plus largement adopté par les chercheurs et les industriels. Dans ce modèle, les droits d'accès sont centrés sur le concept de *rôle* qui représente une fonction dans le cadre d'une organisation. L'accès aux données est accordé à un utilisateur en fonction du rôle qui lui est associé. Cette structuration facilite et simplifie la gestion des permissions.

Une politique de sécurité basée sur le modèle RBAC (une politique RBAC) est généralement intégrée dans les phases de mise en œuvre et d'administration (Lodderstedt et al., 2002). Cela peut rendre le système instable. Les politiques RBAC devraient être intégrées durant la phase de conception, de sorte que ces politiques puissent être spécifiées dès les premières parties de processus de développement. Bien que la nécessité de spécifier une politique de sécurité au niveau des processus métiers ait été soulignée à plusieurs reprises dans la recherche et la pratique, les langages de modélisation des processus métiers ne fournissent pas les éléments nécessaires (Rodríguez et al, 2011). Plusieurs travaux, comme UMLsec (Jürjens, 2004) et SecureBPMN (Brucker et al., 2012), ont proposé de nouveaux langages. Cependant ces langages restent incapables de maîtriser la complexité de politiques RBAC et les risques inhérents aux SI. Il semble donc nécessaire de proposer de nouveaux concepts et de présenter des modèles tenant compte de ces politiques de sécurité.

Notre étude s'intéresse à cette problématique; nous proposons une nouvelle approche, basée sur le langage UML, qui permet de représenter les processus métiers d'un SI en tenant compte d'une politique RBAC.

UML (UML2, 2011) est un langage standard pour la modélisation orientée-objet. Ce langage est devenu une des notations les plus fréquemment utilisées pour l'analyse et la conception des SI. La version 2.0 d'UML fournit 13 types de diagrammes permettant la modélisation graphique des différentes vues statiques (en utilisant le diagramme de classes par exemple) et dynamique (en utilisant le diagramme d'activités par exemple) d'un système. Le langage UML a été proposé comme une notation générique qui permet de spécifier, construire et documenter des systèmes dans des contextes différents et variés tels que le transport, la finance, les télécommunications, etc. Cependant, UML peut ne pas représenter la solution

adéquate pour la modélisation de certains systèmes ou certains aspects d'un système. Le mécanisme de *Profil,* proposé par l'OMG, permet de répondre à ce besoin de spécialisation d'UML. Un profil permet d'augmenter la capacité d'expression des diagrammes d'UML pour modéliser des concepts spécifiques d'un système. Il définit un ensemble de mécanismes d'extension, tels que les stéréotypes, les étiquettes et les contraintes, pour la spécialisation des éléments d'UML. Plusieurs profils sont disponibles pour des domaines variés, comme par exemple, SysML pour les systèmes complexes et MARTE pour les systèmes de temps réel embarqués.

SecureUML (Basin et al., 2009) est un profil UML qui permet de spécifier une politique de contrôle d'accès RBAC au travers des diagrammes d'UML. Dans le cadre de ce profil, une extension des diagrammes de classes d'UML a été proposée pour exprimer le contrôle d'accès aux données du système. Cette extension introduit des permissions pour spécifier les droits d'accès des utilisateurs, affectés à des rôles, aux éléments du diagramme de classes tels que les attributs, les opérations, et les associations. Le profil SecureUML permet également d'associer des contraintes contextuelles, appelées aussi contraintes d'autorisation, aux permissions. Ces contraintes dépendent, d'une part, des utilisateurs et de leurs rôles et, d'autre part, de l'état du diagramme de classes fonctionnel. Elles permettent de spécifier des politiques RBAC qui dépendent des aspects dynamiques du système tels que les valeurs d'attributs et les liens entre les instances des classes.

Le présent travail propose de compléter la vue statique d'une politique RBAC, exprimée au travers des diagrammes de classes SecureUML, par des *diagrammes d'activités d'UML* 2 (UML2, 2011) qui sont l'un des modèles préconisés pour la modélisation des activités d'un processus métier. Nous allons étendre ces diagrammes pour qu'ils représentent le déroulement d'un processus métier en tenant compte d'une politique RBAC et des contraintes contextuelles d'autorisation. Les modèles proposés seront également spécifiés et validés formellement.

I.2 Objectifs et contributions

Cette thèse présente une approche pour la spécification et la validation de politiques RBAC au niveau des activités d'un processus métier. Notre approche poursuit les objectifs suivants :

O1) Définir un profil UML qui adapte le diagramme d'activités d'UML2 pour la spécification de politiques RBAC au niveau des activités d'un processus métier.

O2) Elaborer un ensemble de règles qui assurent la cohérence entre les diagrammes d'activités de contrôle d'accès qui permettent la spécification de la

vue dynamique d'une politique RBAC et les modèles SecureUML qui spécifient la vue statique d'une politique RBAC.

O3) Spécifier et valider formellement la politique RBAC exprimée par les diagrammes d'activités de contrôle d'accès.

Les sections I.2.1, I.2.2 et I.2.3 expliquent les trois phases de notre approche qui permettent respectivement de répondre aux objectifs *O1*, *O2* et *O3*.

I.2.1 Modélisation de politiques RBAC

Le diagramme de la figure I.1 présente les principales étapes de la première phase. Dans cette phase, nous proposons le profil BAAC@UML (Business Activity Access Control with UML) qui étend les diagrammes d'activités d'UML2 pour qu'ils décrivent un processus métier en tenant compte d'une politique RBAC. Le diagramme d'activités d'UML2 fournit plusieurs éléments, tels que les activités, les actions et le flot de contrôle, permettant la description des activités d'un BP.

Notre approche de modélisation des processus métiers commence à partir des cas d'utilisation. Chaque cas d'utilisation encapsule un comportement qui représente des interactions entre le système et ses acteurs. Le profil BAAC@UML détaille ces interactions en tenant compte d'une politique RBAC. En effet, la spécification des modèles autour des cas d'utilisation, qui correspondent aux besoins des utilisateurs, est une bonne pratique dans le développement des systèmes. Dans la figure I.1, nous définissons deux niveaux pour visualiser le comportement d'un cas d'utilisation.

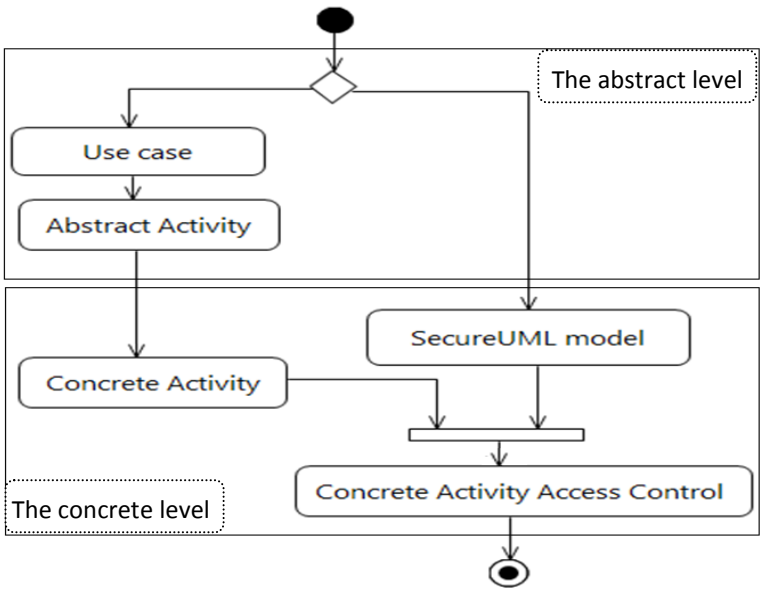

Figure I.1. Démarche de contrôle d'accès aux activités

Le niveau abstrait spécifie un cas d'utilisation par une coordination d'actions abstraites, appelées tâches, qui peuvent être réalisées par ses acteurs. *Le niveau concret* raffine l'abstrait en spécifiant chacune de ses tâches par une coordination d'actions concrètes qui font référence aux opérations des classes fonctionnelles. Ce niveau considère les permissions SecureUML. Notre profil introduit les concepts de *rôle* et *précondition de contrôle d'accès* pour garder l'accès des utilisateurs aux activités concrètes d'un processus métier. Les préconditions de contrôle d'accès font référence aux contraintes contextuelles associées aux permissions SecureUML. Dans cette phase, nous avons défini un méta-modèle associé à des contraintes OCL (OCL, 2012) pour spécifier la sémantique de nos extensions de diagrammes d'activités. Notre méta-modèle a été supporté par l'outil *Graphical-ACP* qui permet la représentation graphique de nos modèles d'activités de contrôle d'accès.

I.2.2 Cohérence de politiques RBAC

Notre profil BAAC@UML permet de spécifier une politique RBAC dans une vue dynamique afin de compléter la spécification statique SecureUML de cette même politique. Cependant, il est nécessaire d'éviter les incohérences entre les deux modèles car ces incohérences peuvent poser beaucoup de problèmes dans la mise en œuvre de la politique de sécurité. Par exemple, un rôle doit avoir les mêmes droits d'accès dans les vues statique et dynamique.

Dans la deuxième phase de notre approche, nous proposons l'élaboration d'un ensemble de règles pour assurer la cohérence structurelle entre les modèles BAAC@UML et les modèles SecureUML. Ces règles vérifient la conformité des rôles et des contraintes contextuelles dans les deux modèles. Cette phase a été supportée par l'outil *ACP-Consistency* qui met en œuvre ces règles de cohérence et permet de montrer les incohérences entre les modèles BAAC@UML et SecureUML.

I.2.3 Spécification et validation formelles de politiques RBAC

Le nombre croissant des risques inhérents aux SI fait que la spécification de politiques RBAC doit suivre une démarche rigoureuse basée sur une sémantique formelle. Cette phase se focalise sur cette question; nous proposons de compléter les modèles BAAC@UML par une spécification formelle exprimée en langage B (Abrial, 1996), et de valider la politique RBAC à partir de cette spécification. La combinaison d'UML et B est motivée par le souhait de pouvoir les utiliser ensemble dans un développement intégrant à la fois structuration et précision.

Dans les activités de vérification et de validation (V&V), nous tirons profit des outils dédiés à la méthode B pour tester la politique RBAC exprimée par les modèles

BAAC@UML et prouver la correction de certaines propriétés de ces modèles et la cohérence sémantique entre ces derniers et les modèles SecureUML.

Dans cette phase, nous proposons une extension de la plateforme B4MSecure (Idani, Ledru, 2015), dédiée à la modélisation en UML et en B des politiques de contrôle d'accès, pour la traduction des modèles BAAC@UML en B.

I.3 Notre approche dans le cadre de l'IDM et MDA

Notre approche s'inscrit dans le cadre de L'*IDM* (Ingénierie Dirigée par les Modèles). L'IDM, ou *Model Driven Engineering* en anglais, est une discipline récente qui se focalise sur les modèles pour le développement des systèmes logiciels. Cette approche a été proposée suite à l'incapacité des approches classiques de développement à répondre aux problèmes liés à la complexité des systèmes. Dans le cadre de l'IDM, l'analyse et la conception d'un système font partie intégrante du processus de développement. Le développement commence par la définition des modèles qui spécifient les différents niveaux d'abstraction du système et facilitent la maîtrise de la complexité et l'automatisation du développement. Le produit final exécutable est considéré comme un élément qui résulte d'une transformation de modèles en exprimant des relations de raffinement entre ces modèles et des règles garantissant leur cohérence. L'IDM offre également le mécanisme standard de méta-modèle qui permet de décrire des modèles et de spécifier leur sémantique. De ce fait, tout modèle doit être conforme à un méta-modèle. En effet, l'IDM cherche à fournir des outils et des techniques permettant aux développeurs de définir des modèles et des méta-modèles mieux adaptés à leurs besoins et d'automatiser les différentes étapes nécessaires à l'élaboration du produit final.

Dans le cadre d'une approche d'IDM, les activités les plus importantes sont la modélisation des différents aspects d'un système, la vérification de la cohérence des modèles et la définition des transformations d'un modèle vers un autre d'une manière automatique. Notre approche aborde les trois activités (voir la figure I.2) ; elle propose les diagrammes BAAC@UML qui modélisent une politique RBAC au niveau des activités d'un processus métier, elle traite de la cohérence des diagrammes BAAC@UML avec les modèles SecureUML, et elle transforme les diagrammes BAAC@UML en B dans le but de vérifier rigoureusement la politique RBAC exprimée au niveau des activités. Ainsi, ces activités d'IDM sont supportées par les outils *Graphical-ACP*, *ACP-Consistency* et *Extended-B4MSecure*.

MDA (Model Driven Architecture) (Miller, Mukerji, 2003) est une approche proposée par l'OMG (Object Management Group)[1] dans le but de supporter l'IDM. Cette approche commence par la définition des modèles d'exigences du système, appelée aussi CIM ou Computation Independent Model. Elle permet ensuite de spécifier séparément des modèles du système indépendants des plateformes spécifiques (PIM ou Platform Independent Model) et des modèles liés aux plateformes (PSM ou Platform Specific Model). L'objectif de cette séparation est de pouvoir déployer un même modèle de type PIM sur plusieurs plateformes (modèles PSM). Dans le cadre de l'approche MDA, l'OMG a défini un standard pour établir de nouveaux langages de modélisation (MOF ou Meta-Object Facility), un standard pour la modélisation (UML ou Unified Modeling Language) et un standard pour modéliser les transformations de modèles (QVT ou *Query, Views, Transformation*).

Notre approche utilise le langage UML préconisé par l'approche MDA. Elle intègre des modèles des niveaux CIM et PIM (voir la figure I.2). Le niveau CIM est représenté par le diagramme de cas d'utilisation qui montre des acteurs interagissant avec les fonctions d'un système. Le niveau PIM est exprimé par les modèles SecureUML et BAAC@UML. Les modèles SecureUML spécifient les données d'un système sous forme de classes et d'associations, et les permissions qui leur sont associées. Les modèles BAAC@UML spécifient le comportement d'un système sous forme d'activités et d'actions ainsi que des préconditions de contrôle d'accès qui leur sont associées. Ils permettent d'établir un pont entre les modèles des cas d'utilisation et les modèles SecureUML.

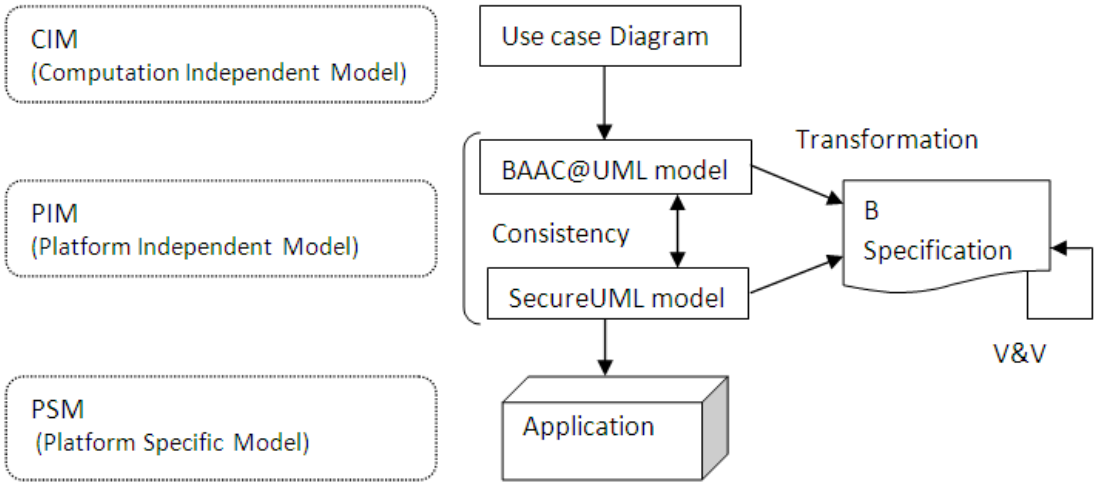

Figure I.2. Notre approche dans le cadre de l'IDM et MDA

[1] *Association américaine créée en 1989 dont l'objectif est de standardiser et promouvoir le modèle objet.*

I.4 Organisation du document

La figure suivante montre l'organisation de cette thèse.

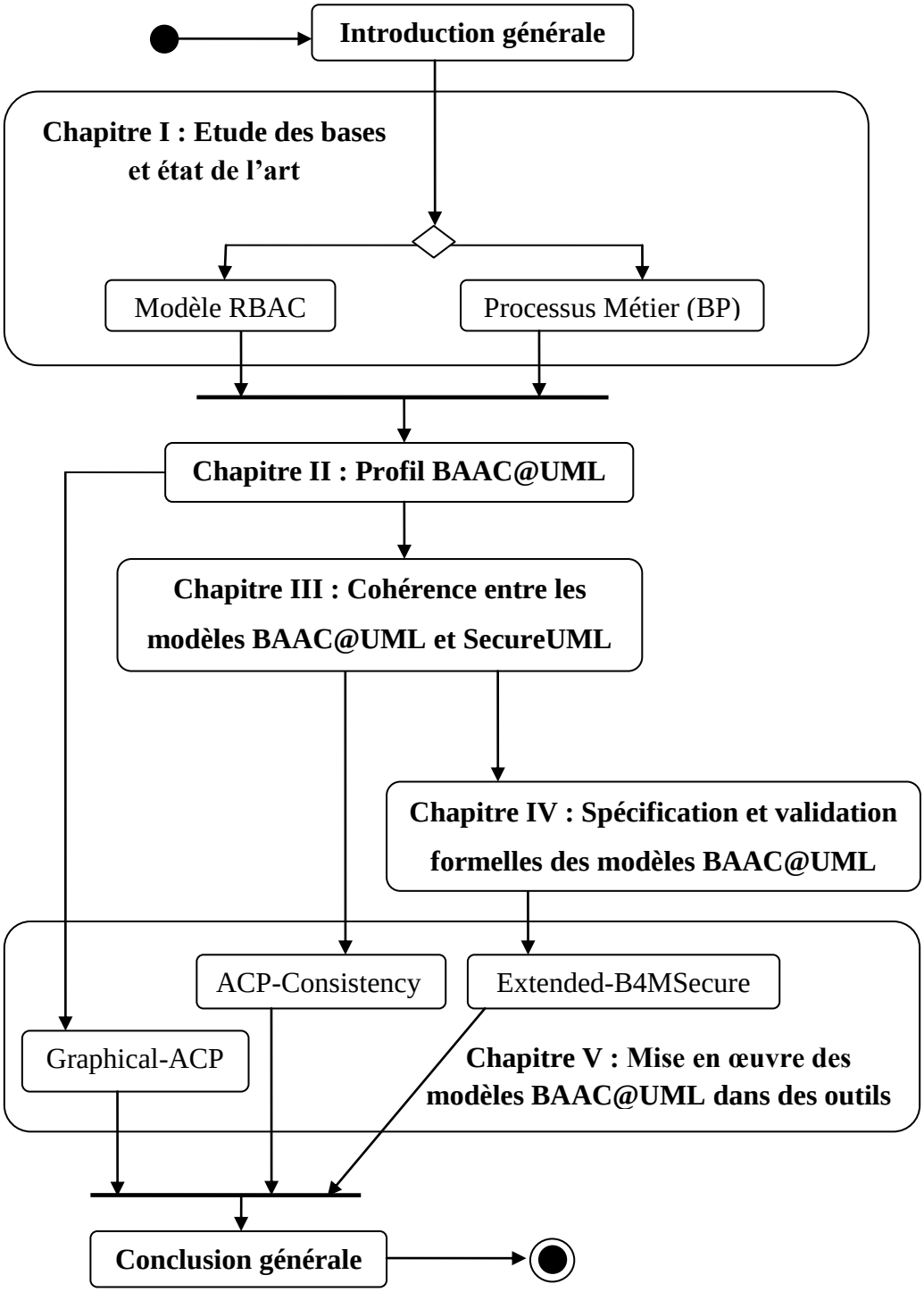

Figure I.3. Organisation de la thèse

Le chapitre 1 définit le Processus Métier (BP), le concept central de notre travail. Il présente également le modèle RBAC utilisé dans notre démarche de spécification de politiques de contrôle d'accès au niveau des activités d'un BP. Un état de l'art des travaux qui combinent RBAC et BP fait également l'objet de ce chapitre.

Dans *le chapitre 2,* nous proposons le profil BAAC@UML qui vise la spécification des activités d'un BP en tenant compte d'une politique de sécurité basée sur le modèle RBAC. Nous décrivons les étapes de spécification des modèles BAAC@UML et un méta-modèle qui définit leur sémantique.

Le chapitre 3 présente un ensemble de règles permettant d'assurer la cohérence entre les modèles de notre profil, qui expriment une politique RBAC dans une vue dynamique, et les modèles SecureUML qui spécifient la vue statique d'une politique RBAC. Une démarche de validation des règles de cohérence est proposée dans ce chapitre.

Dans le chapitre 4, nous proposons une approche de traduction des modèles BAAC@UML en langage B à l'aide de la plateforme B4MSecure. Ce chapitre présente également les activités de vérification et de validation de la politique de sécurité exprimée par les modèles BAAC@UML au moyen des outils dédiés à la méthode B.

Le chapitre 5 explique les étapes de la mise en œuvre de l'outil *Graphical-ACP* qui permet la représentation graphique des modèles BAAC@UML et SecureUML, et l'outil *ACP-Consistency* qui vise à montrer les incohérences entre les deux modèles. Nous discutons également de la mise en œuvre des extensions de la plateforme B4MSecure pour la traduction des modèles BAAC@UML en langage B.

Chapitre 1

Etude des bases et état de l'art

1.1 Introduction

Cette thèse propose le profil BAAC@UML (Business Activity Access Control with UML) qui augmente la capacité d'expression des diagrammes d'activités d'UML2 pour la spécification et la validation de politiques de contrôle d'accès basées sur le modèle RBAC au niveau des activités d'un processus métier.

Pour la compréhension des travaux développés dans cette thèse, ce chapitre met l'accent sur le contexte et les notions de base utilisées dans ce mémoire. Il est divisé en deux parties pour la présentation des deux technologies qui constituent les fondements théoriques et techniques de notre travail : *le processus métier* et *le modèle RBAC*. Nous présentons également un état de l'art des travaux connexes à notre étude en positionnant notre approche parmi ces derniers.

La section 1.2 présente les notions autour du BP, les langages de modélisation des BPs et les travaux qui étendent ces langages pour tenir compte des différentes exigences de sécurité. La section 1.3 définit le modèle RBAC et ses éléments ainsi que les travaux qui traitent de la spécification et de la validation formelle de politiques RBAC au niveau des BPs. La dernière section conclut le chapitre.

1.2 Processus métier (BP)

La notion de BP est centrale dans notre thèse. Dans une organisation, un produit est en général le résultat d'une collaboration de plusieurs acteurs. Le BP est un enchaînement d'activités réalisées par ces acteurs dans le but de délivrer un objectif métier pour l'organisation. Une brève description des concepts et notations du BP est présentée dans la section 1.2.1.

La modélisation des activités d'un BP est l'abstraction de la façon dont collaborent des systèmes et des individus pour réaliser un objectif métier. Elle permet notamment de mieux comprendre le déroulement d'un BP. Plusieurs langages graphiques sont utilisés pour la modélisation des BPs. Cependant BPMN (BPMN2, 2011) et UML (UML2, 2011) sont considérés comme les principales normes (Rodríguez et al., 2007). (Geambasu, 2012) compare BPMN et le diagramme d'activités d'UML2 et conclut qu'ils sont équivalents pour représenter des processus de façon compréhensible et pour décrire des BPs. Dans la section 1.2.2, nous donnons un aperçu du langage BPMN et de principaux éléments du diagramme d'activités que nous allons utiliser dans notre approche.

La spécification de SBP (Secure Business Process) est une nouvelle approche qui considère les aspects de sécurité liés aux BPs durant les premières phases d'analyse

des besoins et de conception. Les différentes exigences de sécurité des BPs et un état de l'art des travaux de spécification de SBP font l'objet de la section 1.2.3.

1.2.1 Concepts et notations

Nous présentons la notion de BP et un méta-modèle qui spécifie ses concepts, la technologie des workflows qui permet d'automatiser les BPs, et enfin l'approche BPM (Business Process Management) qui définit clairement les BPs et leur cycle de vie.

1.2.1.1 Notion de processus métier

Différentes définitions de la notion de BP ont été énoncées dans la littérature. Nous présentons certaines définitions proposées par des organismes de standardisation tels que le WfMC[2] et ISO[3].

Selon (WfMC, 1999), un BP est défini comme étant : « *un ensemble de procédures et d'activités plus ou moins liées qui réalisent collectivement un objectif métier, en général au sein d'une structure organisationnelle définissant des rôles et des relations fonctionnelles*».

D'après (ISO/DIS 19440.2, 2007), *« le processus métier représente un ensemble partiellement ordonné d'activités d'entreprise exécuté pour réaliser des objectifs d'entreprise dans le but d'obtenir un résultat désiré* ».

Le méta-modèle de la figure 1.1 proposé par (Morley et al., 2007) permet de structurer les différents concepts du BP. Le BP est défini par un ensemble d'activités qui participent à l'accomplissement d'un but ou objectif de l'entreprise. Il peut être décomposé en regroupements cohérents d'activités. L'activité est le concept central du BP qui représente des interactions entre différents acteurs sous la forme d'échange d'informations, réalisant des objectifs métiers. Les activités d'un BP sont exécutées par des acteurs consommant et produisant des ressources. Elles peuvent être déclenchées par des événements et peuvent à leur tour produire des événements. Chaque activité représente une collection de tâches organisées dans le temps qui transforme des éléments d'entrée en éléments de sortie. Les tâches décrivent la façon d'effectuer une activité. Elles correspondent à un ensemble d'actions qui, une fois achevées, permettront d'atteindre un résultat précis et mesurable.

[2] *Organisation fondée en 1993 dans le but de promouvoir la technologie des workflows et d'établir des standards pour les systèmes de gestion de workflow.*

[3] *Organisme de normalisation international fondé en 1947. Il est composé de représentants d'organisations nationales de normalisation de 165 pays.*

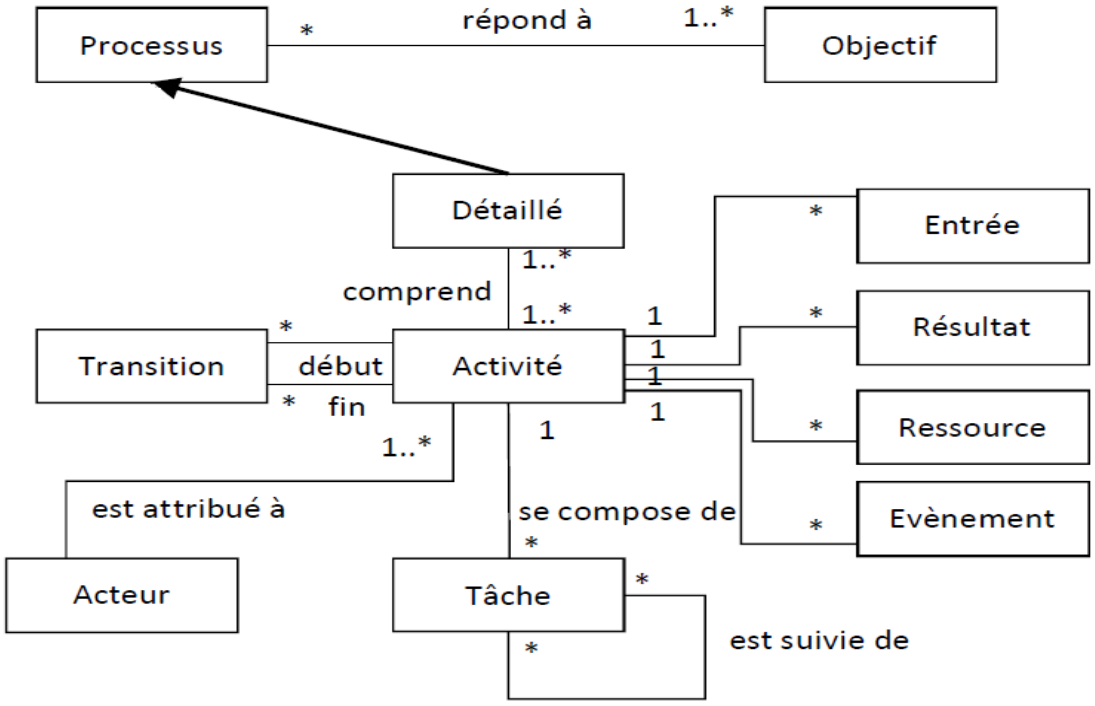

Figure 1.1. Concepts du BP (Morley et al, 2007)

La section suivante présente la technologie des Workflows qui permet de supporter et d'automatiser les BPs.

1.2.1.2 Workflow

Le workflow[4] est défini dans (WfMC, 1999) comme étant « *l'automatisation totale ou partielle d'un processus métier, au cours duquel on échange d'un participant à un autre, des documents, des informations ou des tâches pour action, et ce selon un ensemble de règles procédurales* ».

La figure 1.2 présente la structuration des différents concepts d'un workflow. Ces concepts sont classifiés selon deux points de vue : *une vue conceptuelle* qui englobe la représentation des BPs et *une vue d'exécution* qui traite l'exécution des instances des BPs et les instances de leurs activités.

La vision conceptuelle (partie gauche de la figure 1.2) représente une arborescence des concepts liés à la représentation d'un BP. Ce dernier peut se décomposer en sous-processus où chacun inclut un ensemble d'activités. Dans cette définition, les activités peuvent être automatiques, exécutées par le système, ou manuelles exécutées par les acteurs humains. La vue conceptuelle décrit le cheminement des activités, les acteurs

[4] *Un workflow est traduit littéralement en français par « flux de travail ».*

du processus, les applications informatiques associées au processus, les données utilisées dans le processus, etc.

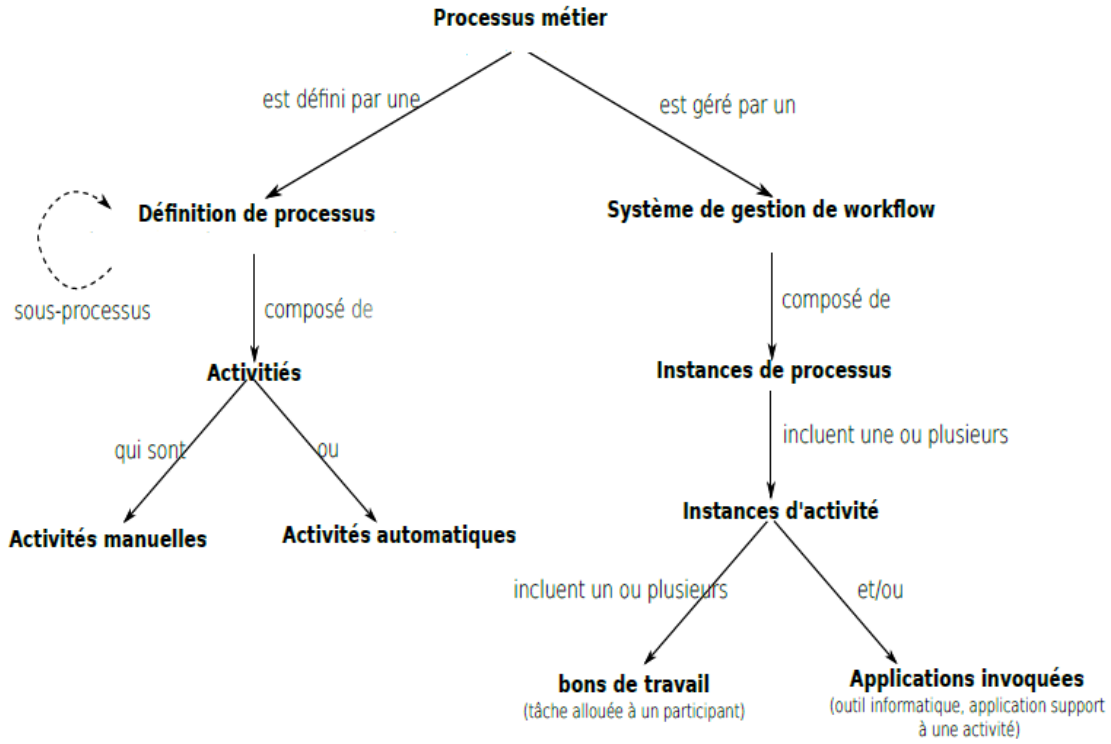

Figure 1.2. Concepts du workflow d'un BP (WFMC, 1999)

La vision d'exécution (partie droite de la figure 1.2) définit les concepts liés à l'exécution du workflow au moyen d'un système de gestion de workflow (WMS (Workflow Management System)). WMS comprend les méthodes, les techniques et les outils utilisés pour gérer l'automatisation et l'exécution des instances d'un BP. Une instance de processus inclut une ou plusieurs instances d'activités qui se présentent comme une liste de tâches à effectuer et/ou d'applications informatiques invoquées.

1.2.1.3 BPM

Plusieurs définitions ont été proposées au terme « *Gestion de Processus Métier* », ou en anglais « *Business Process Management* (BPM) ». Nous avons choisi la définition de (Ulmer, 2011) qui est un consensus des différentes définitions. Selon (Ulmer, 2011) : « *BPM correspond à l'ingénierie des processus de l'organisation, ou processus métier, à l'aide des technologies de l'information. Il a pour vocation de modéliser, déployer, exécuter et optimiser de manière continue les différents types de processus et ainsi d'améliorer l'agilité d'une organisation* ».

BPM présente une véritable architecture qui fournit une réponse adaptée aux différents besoins d'utilisateurs, en abordant la question de développement des systèmes priorisant le point de vue « métier » par rapport au point de vue technique. L'objectif du BPM est de :

- Permettre aux entreprises de gérer les BPs depuis un niveau plus haut jusqu'à un niveau opérationnel.
- Améliorer la modélisation et l'intégration des BPs et également l'augmentation de la productivité et la diminution des coûts (Ulmer, 2011).
- Garantir à une entreprise que ses BPs sont adaptés de manière continue à un environnement en constante évolution (Fingar, Bellini, 2004).
- Offrir aux organisations la liberté de changer rapidement leurs systèmes et leurs processus, sans avoir besoin de redévelopper complètement leurs applications.

BPM repose sur un ensemble de langages, méthodes, outils et standards prenant en charge le cycle de vie d'un BP depuis la conception jusqu'à l'implémentation. Différentes visions ont été proposées au sujet du cycle de vie de gestion d'un BP, comme par exemple (Aalst, 2004) et (Muehlen, 2004). (Aalst, 2004) est la plus répandue (Zefouni, 2012). Cette vision présente le cycle de vie sous forme d'un anneau signifiant que la démarche est dans une optique d'amélioration continue. La figure 1.3 montre cette représentation qui comprend quatre phases.

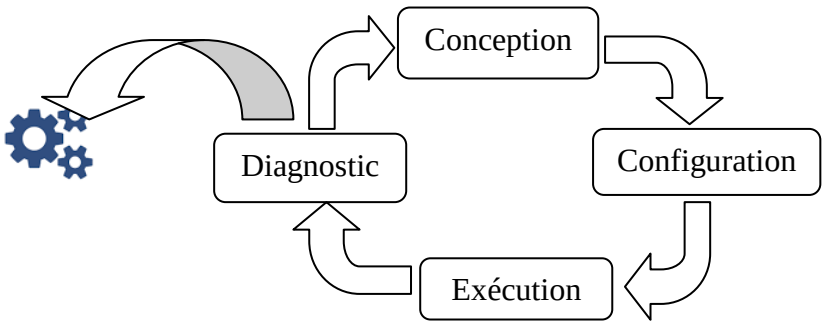

Figure 1.3. Cycle de vie d'un BP

La phase de conception a pour objectif la modélisation des BPs. La modélisation se fait à l'aide des notations, souvent graphiques, qui décrivent les tâches menées dans le cadre d'un BP, l'enchaînement de ces tâches et les acteurs impliqués dans le processus. Il est également possible, dans cette phase, de valider les modèles de BP à l'aide des outils de simulation et de vérification. Dans le cadre de notre étude, nous nous sommes positionnés dans cette phase ; nous proposons une approche de modélisation des BPs qui tient compte d'une politique de sécurité. La section suivante présente les langages qui servent de support à cette phase.

La phase de configuration produit une version exécutable du BP. Elle consiste à développer les tâches de BP modélisé et à connecter ces tâches au SI (accès effectifs des utilisateurs aux données). Les paramètres d'exécution, le format des messages échangés, les protocoles de transports utilisés et les droits des utilisateurs sont aussi définis dans cette phase. La version exécutable peut être testée avant le déploiement final.

La phase d'exécution consiste à exécuter le BP en utilisant la version définie dans la phase précédente. L'exécution est réalisée, en général, au moyen d'un moteur qui soumet les tâches aux différents acteurs et exécute les instances de processus conformément à leur modèle.

La phase de diagnostic consiste à analyser les traces d'exécution de processus pour mesurer l'écart entre sa version exécutable et son modèle. Cette phase peut amener à revenir sur l'étape de conception pour corriger les modèles. Il est également possible de suivre les indicateurs de performances, comme par exemple la vitesse d'exécution, lors de l'exécution du processus. En effet, le but de cette phase est d'améliorer les performances et de vérifier la conformité de la version exécutable au modèle de processus.

1.2.2 Langages de modélisation des BPs

La modélisation des BPs joue un rôle fondamental car elle permet d'obtenir une représentation graphique décrivant les BPs et facilitant leur compréhension. Pour cela, des langages doivent être utilisés pour permettre l'élaboration des modèles qui spécifient les différentes activités du BP. En effet, le langage de modélisation du BP véhicule le fonctionnement du processus en utilisant une syntaxe qui détermine la bonne construction des expressions représentant les éléments du processus et une sémantique qui détermine la manière dont les expressions du langage doivent être interprétées (Boukhebouze, 2010). La littérature distingue deux catégories de langages de modélisation des BPs : *les langages impératifs* et *les langages déclaratifs*.

Dans les *langages déclaratifs*, un processus est vu comme un ensemble d'états et un ensemble de contraintes qui contrôlent les transitions entre les différents états. Des paradigmes, comme la logique temporelle linéaire et la modélisation basée sur les règles, sont utilisés pour représenter les états et les contraintes. Parmi les langages déclaratifs, on peut citer : *ConDec* (Pesic, van der Aalst, 2006), *PLM-flow* (Zeng et al., 2002) et *PENELOPE* (Goedertier, Vanthienen, 2006).

Dans les *langages impératifs*, la modélisation des processus se focalise sur la définition de la manière dont les différentes activités sont réalisées. Pour cela,

l'enchainement dans l'exécution des activités doit être décrit d'une façon explicite en utilisant des liens ou des connecteurs. Parmi les langages impératifs, on peut citer: *BPMN* (*BPMN2*, 2011), *Diagramme d'activités d'UML2* (UML2, 2011) et *EPC* (Event-driven Process Chain) (Davis, Brabander, 2007).

Dans le cadre de notre travail, nous nous sommes intéressés à l'approche impérative. Actuellement, les langages impératifs les plus reconnus sont : le diagramme d'activités d'UML2 et la notation BPMN. Notre choix, dans le cadre de cette étude, porte sur les diagrammes d'activités d'UML2. Il s'agit d'une notation graphique permettant de décrire tous les aspects d'un BP à l'aide d'un seul type de diagramme. Ainsi, le mécanisme de profil d'UML permet l'extension de ces diagrammes pour tenir compte d'un aspect particulier du système, comme l'aspect de sécurité.

1.2.2.1 BPMN

BPMN (Business Process Modeling Notation) est une notation graphique dédiée à la modélisation des BPs. Elle a été proposée initialement par le consortium BPMI[5] (Business Process Management Initiative) en 2004. Depuis 2005, OMG[6] (Object Management Group) a repris la maintenance et l'évolution de la notation. La spécification des BPs à l'aide de BPMN se fait en deux phases :

- La première phase est consacrée à l'élaboration des modèles de BPs sans tenir compte des aspects techniques et d'exécution des processus modélisés.
- La seconde partie décrit la traduction de différents diagrammes BPMN en éléments du langage BPEL (Business Process Executable Language) d'exécution des BPs.

BPMN propose le BPD (Business Process Diagram) pour la spécification des BPs. La notation BPMN couvre uniquement la description des éléments de spécification des BPDs sans préciser de méthodologie particulière pour leur spécification.

1.2.2.2 Le diagramme d'activités d'UML2

Le diagramme d'activités est un modèle UML couramment utilisé pour représenter graphiquement les aspects dynamiques des systèmes et les activités d'un processus. La version 2.x de ces diagrammes a été revue pour être mieux adaptée à la modélisation des BPs (Russell, 2006) (Méga, 2008). Il existe plusieurs utilisations courantes de diagrammes d'activités. Par exemple, (Audibert, 2008) affirme que les diagrammes

[5] *Regroupe des entreprises leaders du marché comme IBM, BEA, Siebel, ... etc.*
[6] *Association américaine créée en 1989 dont l'objectif est de standardiser et promouvoir le modèle objet.*

d'activités sont particulièrement adaptés à la description des cas d'utilisation et ils viennent illustrer et consolider la description textuelle des cas d'utilisation. Il considère ainsi qu'on peut attacher un diagramme d'activités à n'importe quel élément de modélisation afin de visualiser, spécifier, construire ou documenter le comportement de cet élément. (Le Thi, 2011) considère que les diagrammes d'activités peuvent intervenir dans un processus IDM en amont des activités de génération des codes car ils offrent une vision très proche de celle des langages de programmation impératifs comme *C++* ou *Java*.

L'élément principal de spécification d'un processus par un diagramme d'activités est l'activité. Une activité est un comportement qui peut être représenté par un enchainement d'unités dont les éléments simples sont les actions. Le flot d'une activité est modélisé par des nœuds reliés par des arcs (transitions). Nous présentons dans la suite de cette section les principaux types d'éléments qui peuvent être contenus dans un diagramme d'activités. Ces éléments sont illustrés dans la figure 1.4. Leurs définitions sont inspirées essentiellement de (UML2, 2011) et (Audibert, 2008).

1.2.2.2.1 Nœuds d'activité

Les nœuds d'activité permettent de représenter les étapes d'une activité d'un processus. Ils peuvent être des nœuds exécutables, des nœuds d'objets ou des nœuds de contrôle.

a) Nœud exécutable : Un nœud exécutable (executable node) est un type d'élément abstrait qui spécifie une action d'une activité qu'on peut exécuter. Il possède un gestionnaire d'exception et est relié à des flots entrants et des flots sortants. Les nœuds exécutables peuvent être simples ou structurés.

Un nœud d'action (action node) est un nœud exécutable simple qui constitue l'unité fondamentale de la spécification du comportement de l'activité. Une action représente, par exemple, une création d'un objet, une affectation d'une valeur à un attribut, l'accès à une valeur d'attribut ou de terminaison d'association, une transformation ou un calcul. Les actions sont généralement liées à des opérations qui sont directement invoquées. Elles peuvent être ainsi reliées à des préconditions en précisant ce qui doit être rempli lorsque l'action est invoquée ou des post-conditions en précisant ce qui doit être accompli après l'exécution de l'action. Dans un diagramme d'activités, on peut trouver une très grande variété de types d'actions. Parmi les actions les plus courantes, on peut citer :

- *call operation* qui correspond à l'invocation d'une opération sur un objet de manière synchrone ou asynchrone.
- *call behavior* qui invoque directement une activité plutôt qu'une opération.

- *opaque* représente une action qui peut contenir du code dans un langage qui devra être précisé.
- *send* qui crée un message et le transmet à un objet cible, où il peut déclencher un comportement.
- *accept event* qui bloque l'exécution en cours jusqu'à la réception d'un événement.
- *create* et *destroy* qui permettent respectivement d'instancier et de détruire un objet.

Un nœud d'activité structuré (structured activity node) est un nœud exécutable composé qui représente une portion structurée d'une activité. Il regroupe des actions semblables aux structures de contrôle des langages de programmation telles que la séquence (sequence node), le test (conditional node) et la boucle (loop node). Chacune de ces actions est reliée, par un lien de composition, au nœud structuré qui l'inclut.

b) Nœud de contrôle : Un nœud de contrôle (control node) est un élément abstrait qui représente des nœuds permettant de coordonner les flots entre les nœuds d'une activité. Il existe plusieurs types de nœuds de contrôle qui servent d'agent de circulation. On peut citer par exemple :

- *initial node* qui représente l'endroit où le flot commence quand une activité est invoquée. Il possède un arc sortant et pas d'arc entrant.
- *final node* qui termine l'exécution de l'activité. Il possède un ou plusieurs arcs entrants et aucun arc sortant.
- *decision node* qui permet de faire un choix entre plusieurs flots sortants. Il relie un flot entrant et plusieurs flots sortants.
- *merge node* qui rassemble plusieurs flots alternatifs entrants en un seul flot sortant. Il accepte un flux (en sortie) parmi plusieurs flux entrants.
- *fork node* qui sépare un flux d'entrée en plusieurs flots concurrents en sortie. Il possède donc un arc entrant et plusieurs arcs sortants.
- *join node* qui synchronise des flots multiples. Il possède plusieurs arcs entrants et un seul arc sortant.

c) Nœud d'objet : Un nœud d'objet (object node) est un type abstrait qui représente l'existence d'un objet généré par une action dans une activité et utilisé par d'autres actions. Parmi les types de nœud d'objet :

- *pin* qui spécifie des valeurs passées en argument à une action et des valeurs de retour. L'action ne peut débuter qu'après l'affectation d'une valeur à chacun de ses pins d'entrée. Après l'exécution de l'action, des valeurs doivent être affectées à chacun de ses pins de sortie.
- *activity parameter node* qui décrit les entrées nécessaires à l'exécution d'une activité ou les sorties d'une activité.

- *central buffer node* qui peut contenir des valeurs en provenance de diverses sources et livrer des valeurs vers différentes destinations. Les flots en provenance d'un nœud tampon central ne sont donc pas directement connectés à des actions.
- *data store node* est un nœud tampon qui assure la persistance des données. L'information est dupliquée et ne disparaît pas du nœud de stockage des données comme ce serait le cas dans un *central buffer node*.

1.2.2.2.2 Transitions d'activité

Une transition d'activité est une connexion dirigée (arc) entre deux nœuds d'activité qui spécifie le passage du nœud source au nœud de destination. Elle spécifie l'enchaînement des activités d'un processus et définit *le flot de contrôle* et *le flot d'objet*.

Un flot de contrôle (control flow) indique l'ordonnancement de l'exécution de deux nœuds d'activité. L'exécution du nœud d'activité sera suivie de l'exécution de ses nœuds successeurs. Tout nœud d'activité peut recevoir plusieurs flots de contrôle entrants, et inversement peut être à l'origine du départ de plusieurs flots de contrôle sortants. Dès qu'un nœud d'activité a terminé son exécution, le contrôle est alors donné au nœud d'activité sortant. Les transitions sont franchies de manière automatique. Elles provoquent automatiquement et immédiatement le début des nœuds cibles après le déclenchement des nœuds sources.

Un flot d'objet (object flow) permet de faire passer des données entre des nœuds d'objet. On distingue deux représentations : la première à travers les pins rattachés aux actions ; elle relie un pin de sortie à un pin d'entrée. Le type du pin récepteur doit être identique du type du pin émetteur. La deuxième est illustrée dans la figure 1.4. Elle fait intervenir un nœud d'objet (comme *Devis* et *Facture*) détaché d'une activité particulière. Des arcs viennent ensuite relier ce nœud d'objet à des activités sources et cibles. Le nom d'un état ou d'une liste d'états de l'objet peut être précisé au niveau du nœud d'objet.

1.2.2.2.3 Partition et exception

Les partitions, appelées aussi couloirs (*swimlanes*), permettent de diviser une activité d'un processus en regroupements de nœuds avec leurs liens. Une partition peut être décomposée en sous-partitions et regrouper d'autres partitions selon une autre dimension (les partitions bidimensionnelles). Les partitions n'ont pas une sémantique précise, mais elles correspondent souvent à des unités organisationnelles dans un modèle de processus. On peut, par exemple, les utiliser pour spécifier les

acteurs responsables de l'exécution du comportement défini par les nœuds inclus dans ladite partition. Dans le cas d'un diagramme d'activités partitionné, les nœuds d'activités appartiennent forcément à une et une seule partition. En revanche, les transitions peuvent traverser les frontières des partitions.

Une exception est générée quand une situation anormale entrave le déroulement nominal d'une activité d'un processus. Elle peut être générée automatiquement pour signaler une erreur d'exécution ou être soulevée explicitement par une action (Raise Exception) pour signaler un cas d'erreur qui n'est pas pris en charge par la séquence normale. Dans le cas d'une exception, l'exécution de l'activité est abandonnée sans générer de valeur de sortie. Le mécanisme d'exécution recherche alors un gestionnaire d'exception susceptible de traiter l'exception.

La figure 1.4, qui modélise un processus de gestion de commandes, illustre l'utilisation des différents éléments du diagramme d'activités.

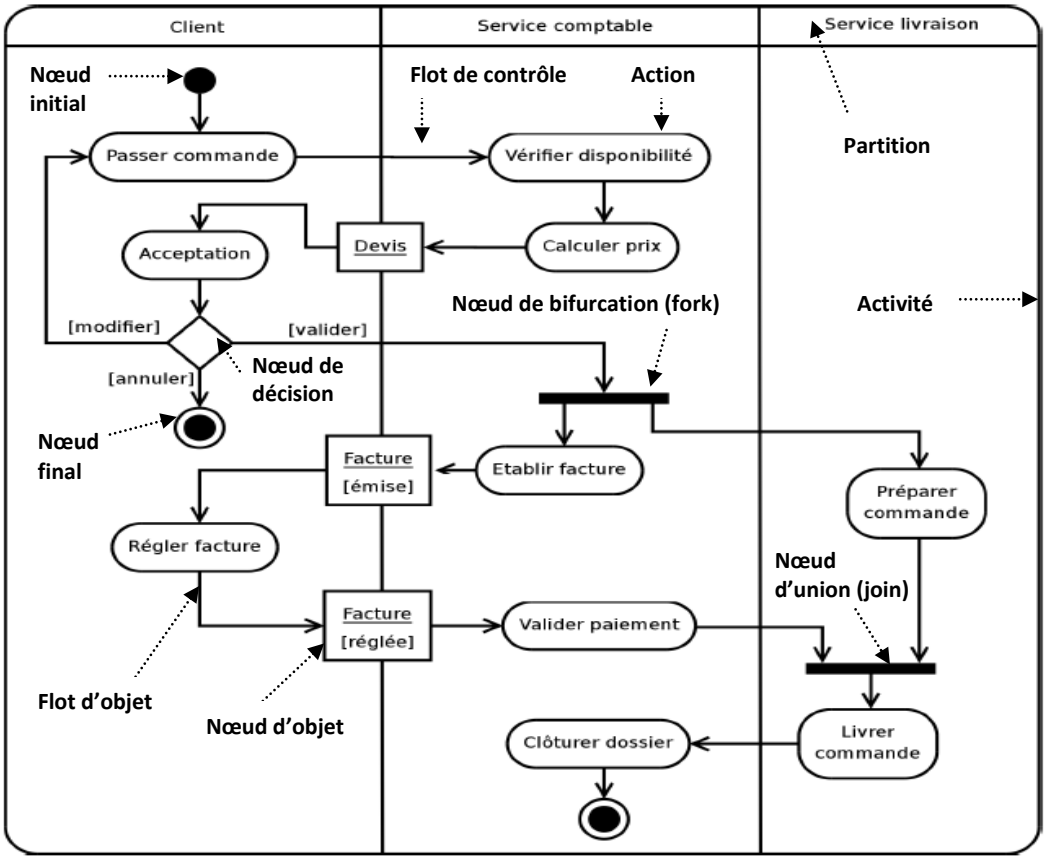

Figure 1.4. Exemple d'un diagramme d'activités (Audibert, 2008)

Le diagramme d'activités fournit donc tous les éléments nécessaires pour décrire les activités d'un BP. Cependant ce diagramme doit être étendu pour qu'il puisse exprimer les aspects de sécurité liés aux BPs. La section suivante présente les travaux qui proposent des extensions de langages de modélisation tels que le diagramme d'activités pour tenir compte des exigences de sécurité dans la spécification des BPs.

1.2.3 Spécification des BPs sécurisés (SBP)

Depuis ces dernières années, la prise en compte des aspects de sécurité dans la spécification des BPs est devenue une préoccupation majeure. Cette approche, appelée aussi spécification des SBP (Secure Business Process), a été proposée pour traiter les problèmes de sécurité liés aux BPs avant leur apparition. Beaucoup de processus deviennent plus mobiles, flexibles, et distribués et sont exécutés dans des contextes différents (Rosemann et al., 2008). Avec cette nouvelle donne et l'augmentation du nombre d'utilisateurs, les BPs sont devenus des cibles pour des attaques internes et externes qui ont beaucoup affecté la confidentialité, l'intégrité et la disponibilité des informations. L'approche de spécification des SBP considère ces exigences de sécurité durant les phases de capture des besoins, d'analyse et de conception.

1.2.3.1 Exigences de sécurité des BPs

Bien que les exigences de sécurité sont largement traitées dans la littérature, il est difficile de trouver une définition suffisamment large pour décrire toutes leurs propriétés (Zuccato, 2004). Dans cette section, nous présentons les exigences de sécurité considérées par (Rodriguez et al, 2011) et qui répondent aux trois caractéristiques : La clarté de leur définition, leur importance dans la sécurité des BPs et leur indépendance par rapport aux solutions de sécurité spécifiques. Ces exigences sont : le contrôle d'accès, la détection des dommages d'attaques, la non-répudiation, l'intégrité, la confidentialité et enfin l'audit de sécurité. Dans le cadre de notre étude, nous nous sommes focalisés sur le contrôle d'accès. Nous allons expliquer les six exigences de sécurité et montrer comment le contrôle d'accès peut assurer certaines de ces exigences.

Le contrôle d'accès limite l'accès aux ressources d'un BP aux seules personnes autorisées. Il est défini dans (WfMC, 2001) comme étant « *la politique par laquelle les utilisateurs sont autorisés à accéder aux différentes opérations ou données dans un système informatique, en fonction de leur identité et des privilèges associés*». Le contrôle d'accès est pris en compte par la plupart des systèmes informatiques comme par exemple les systèmes d'information et les systèmes d'exploitation. Il est exprimé

par un ensemble de règles qui définissent qui peut accéder à quelles actions ou tâches et sous quelles conditions.

La détection des attaques est la mesure par laquelle une tentative d'attaque ou une attaque réussie est inscrite et notifiée (Rodriguez et al, 2011). Il existe des outils au niveau technique qui permettent la détection des attaques et la traçabilité des actions malveillantes. Dans la section suivante nous présentons des exemples de travaux qui spécifient des cas d'attaques au niveau des activités d'un BP.

La non-répudiation assure que chaque partie d'une interaction, comme par exemple, un message, transaction, ou la transmission de données est identifiée de manière unique. Les signatures et les certificats numériques sont utilisés pour vérifier l'identité de l'expéditeur et fournir la preuve d'une interaction. Dans un contexte de workflow, la non-répudiation est particulièrement importante lorsqu'il s'agit de systèmes critiques (WfMC, 2001).

L'intégrité, selon la norme ISO 7498-2, est « *la prévention d'une modification non autorisée de l'information* ». Elle consiste à protéger les BPs contre la corruption intentionnelle (conme par exemple, la création, la modification ou la suppression non autorisée de données). Des données, si elles ont été modifiées de façon illégale, peuvent devenir dangereuses. Dans le contexte d'un BP, l'intégrité est une propriété très importante. Le contrôle d'accès contribue à la vérification de l'intégrité au niveau d'un BP en limitant l'accès aux tâches manipulant les données pour utilisateurs légitimes seulement.

La confidentialité, d'après la norme ISO 7498-2, est « *la propriété qu'une information n'est ni disponible ni divulguée aux personnes, entités ou processus non autorisés* ». Les utilisateurs ne devraient avoir accès qu'aux données nécessaires à l'exercice de leur activités. Les degrés de confidentialité dépendent de la nature des informations et de ceux qui vont les utiliser. Dans le cadre d'un BP, il est important d'assurer la confidentialité des informations. Une politique de contrôle d'accès doit être mise en place pour s'assurer que seules les personnes ayant besoin d'accéder à certaines informations puissent y accéder.

L'audit, selon (WfMC, 2001), consiste à conserver un historique des événements du système et des opérations à travers le système informatique pour permettre l'identification ultérieure des événements d'intérêt ou notamment des évènements liés à la sécurité. Il permet d'assurer une traçabilité des activités menées à travers l'exécution des différentes activités d'un BP. Cela permet la facturation correcte des actes réalisés par tous les utilisateurs. La traçabilité permet d'éviter des abus en matière d'accès abusif aux données.

1.2.3.2 Approches de spécification des SBP

Plusieurs approches de spécification des SBP ont été proposées dans la littérature. Ces approches proposent des extensions de langages de modélisation des BPs pour spécifier les besoins de sécurité, les attaques possibles sur les BPs et des politiques de contrôle d'accès au niveau des BPs. Dans la suite de cette section, nous présentons les principaux travaux basés sur le diagramme d'activités d'UML et le langage BPMN.

L'approche de spécification de besoins de sécurité propose des extensions des langages de modélisation pour représenter les différents services de sécurité qui doivent être pris en compte dans le développement d'un SBP. Cette approche s'intéresse aux exigences de sécurité de haut niveau et se situe en grande partie au niveau de l'activité d'analyse des besoins. Les travaux de *Rodriguez,* par exemple, sont classés dans cette catégorie. (Rodriguez et al, 2011) présente une extension de diagrammes d'activités d'UML2 (nommée BPSec) pour spécifier les différentes exigences de sécurité, telles que la confidentialité, l'intégrité, au niveau des BPs. BPSec est basée sur l'approche MDA. Elle permet de spécifier des modèles de niveau CIM qui sont un point de départ pour obtenir des modèles de sécurité de niveau PIM comme par exemple des diagrammes de cas d'utilisation et des diagrammes de classes. (Rodriguez et al, 2007) étend le méta-modèle du langage BPMN par l'intégration des éléments permettant la spécification des exigences de sécurité à travers les diagrammes BPD. Les exigences de sécurité seront ensuite raffinées par des experts en sécurité et intégrées dans le développement du produit logiciel final.

L'approche de spécification des attaques étend les langages de modélisation pour spécifier des scénarios d'attaque sur les BPs. Le but de cette approche n'est pas de modéliser des BPs sécurisés mais plutôt de modéliser les menaces possibles sur ces processus. (Sindre, 2007), par exemple, est inscrit dans cette approche. Ce travail propose une extension du diagramme d'activités (Mal-Activity Diagram (MAD)) pour modéliser les activités malveillantes d'un BP. MAD utilise la même syntaxe et sémantique du diagramme d'activités d'UML, mais il ajoute des nouveaux concepts comme l'activité malveillante, l'acteur malveillant et la décision (Nœud de contrôle) malveillante (prise dans un but malveillant). Les trois concepts sont représentés avec les mêmes icônes, mais avec une couleur de remplissage noir. L'approche de *Sindre* propose la modélisation des activités malveillantes ainsi que des activités légitimes dans le même modèle.

L'approche de spécification de politiques de contrôle d'accès propose des extensions des langages de modélisation des BPs pour exprimer des règles d'une politique de sécurité basée sur l'un des modèles de contrôle d'accès tels que RBAC (Role Based Access Control) et ABAC (Attribute Based Access Control) (Yuan,

Tong, 2005). Cette approche cible les activités de conception et de mise en œuvre avec un niveau de détail plus important. (Ayoub et al., 2012), par exemple, étend BPMN pour spécifier une politique de sécurité basée sur le modèle ABAC. Plusieurs approches ont été proposées pour la spécification de politiques RBAC. Parmi ces approches: (Brucker et al., 2012) propose SecureBPMN qui étend BPMN. (Jürjens, 2004) et (Strembeck, Mendling, 2011) proposent des extensions de diagrammes d'activités. Ces deux travaux seront présentés avec plus de détails dans la section 1.3.4.

Notre travail porte sur la dernière approche. Nous proposons des extensions des diagrammes d'activités d'UML 2 pour la spécification d'une politique RBAC. La principale différence entre notre approche et les autres approches basées sur RBAC est qu'elle intègre des contraintes contextuelles d'autorisation qui font référence aux informations issues du modèle de sécurité (comme l'utilisateur et ses rôles) et à l'état du modèle fonctionnel (comme les valeurs d'attributs). Notre approche permet également de traiter la cohérence entre les diagrammes d'activités de contrôle d'accès et les modèles statiques de politique RBAC (spécifiés par le profil SecureUML) et de spécifier et valider formellement les diagrammes d'activités de contrôle d'accès.

On pourrait dire que notre approche et les deux autres approches de spécification des SBPs sont complémentaires, puisque chacune est axée sur un problème différent. Notre approche spécifie une politique de sécurité répondant aux différentes exigences de sécurité (qui peuvent être exprimées par BPSec par exemple). Les modèles MAD peuvent également compléter notre approche par la spécification des attaques possibles sur les activités d'un BP.

Dans la section suivante, nous présentons le modèle RBAC utilisé dans notre approche de contrôle d'accès au niveau des activités d'un BP.

1.3 Le modèle RBAC

L'augmentation des risques inhérents aux systèmes et la complexité croissante de politiques de sécurité mises en œuvre, ont conduit à l'émergence de plusieurs modèles de contrôle d'accès, comme par exemple, le modèle contrôle d'accès discrétionnaire (Discretionary Access Control ou DAC) (Butler, Lampson, 1971), le modèle de contrôle d'accès obligatoire (Mandatory Access Control ou MAC) (Bell, LaPadula, 1973), le modèle de contrôle d'accès à base de tâches (Task Based Access Control ou TBAC) (Thomas, Sandhu, 1993), le modèle de contrôle d'accès à base d'organisation (Organization Based Access Control ou OrBAC) (Abou El Kalam et al., 2003), le modèles de contrôle d'accès à base de rôle (Role Based Access Control ou RBAC) (Ferraiolo et al., 2003) et autres.

Notre approche de contrôle d'accès aux activités d'un BP est basée sur le modèle RBAC. Ce modèle est supporté actuellement par plusieurs plateformes et technologies qui le qualifient comme standard de contrôle d'accès. Dans RBAC, chaque décision d'accès est basée sur le rôle assigné à l'utilisateur. Pour chaque accès au système, un utilisateur est assigné à un ensemble de rôles qui lui donnent des permissions pour accéder à un ensemble d'objets.

Le modèle RBAC est organisé en trois niveaux (voir tableau 1.1). Le premier, appelé *Flat RBAC*, définit les concepts de base qui représentent le noyau du modèle. Le deuxième, appelé *Hierarchical RBAC*, introduit la relation hiérarchique entre les rôles qui permet l'héritage des permissions assignées aux rôles. Le troisième, appelé *Constrained RBAC*, introduit les notions de séparation de devoirs statique et dynamique. Dans les sections 1.3.1 et 1.3.2, nous proposons de décrire brièvement les trois niveaux. La section 1.3.3 présente les travaux de spécification de politiques RBAC au niveau des BPs. La section 1.3.4 décrit les travaux qui étendent les diagrammes d'activités pour la spécification de politiques RBAC. Enfin, la section 1.3.5 introduit les travaux qui traitent la validation formelle de politiques RBAC au niveau des BPs.

RBAC 0	Noyau RBAC (Flat RBAC)
RBAC 1	RBAC avec hiérarchie de rôles (Hierarchical RBAC)
RBAC 2	RBAC avec contrainte sur les rôles (Constrained RBAC)

Tableau 1.1. Les trois couches du modèle RBAC

1.3.1 Eléments de base

La figure 1.5 schématise les éléments et les relations du modèle RBAC de référence fourni par (ANSI, 2004). Le niveau *Flat RBAC* est défini par les concepts utilisateur (USERS), rôle (ROLES), session (SESSIONS), permission (PRMS), opération (OPS) et objet (OBS) ainsi que les relations PA (Permission Assignment), UA (User Assignment), session_users et session_roles.

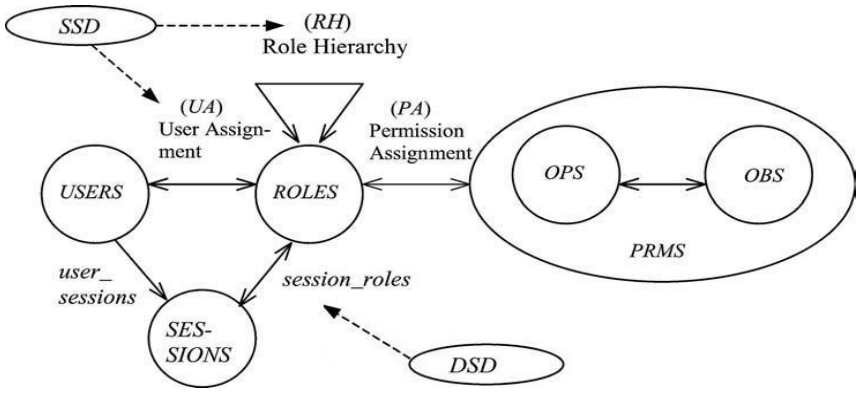

Figure 1.5. Schéma du modèle RBAC (ANSI, 2004)

USERS représente l'ensemble des agents qui vont interagir avec le système. Bien que le concept d'utilisateur puisse être étendu pour inclure des machines, des réseaux ou des systèmes logiciels, il est généralement assimilé aux utilisateurs humains qui accèdent aux ressources de système.

OBS représente l'ensemble des entités du système susceptibles de faire l'objet d'accès et qui nécessitent une protection. Une entité représente toute ressource qui contient ou reçoit des informations. Elle peut représenter des conteneurs d'informations (comme, par exemple, des fichiers et des répertoires dans un système d'exploitation ou des tables et des enregistrements de bases de données) ou des ressources épuisables telles que des imprimantes et des disques.

OPS représente l'ensemble des actions qui récupèrent ou communiquent des informations aux objets du système. Une opération représente les types d'accès aux objets du système. Ces types d'opérations dépendent du type de système. Par exemple, dans un système de fichiers, les opérations peuvent être lire, écrire ou exécuter. Dans un système de gestion de base de données, les opérations peuvent être insérer, supprimer, ajouter ou mettre à jour.

PRMS représente l'ensemble des privilèges qui autorisent la réalisation d'actions sur les ressources protégées du système. Une permission permet de faire le lien entre des opérations et des objets (PRMS $\subseteq$ OPS $\times$ OBS).

ROLES représente l'ensemble des fonctions ou des responsabilités identifiées dans l'organisation. Le concept central de rôle permet la construction d'un pont entre les utilisateurs et les permissions à travers les relations UA et PA. UA est une relation qui permet d'affecter des utilisateurs aux rôles (UA $\subseteq$ USERS $\times$ ROLES). Un même utilisateur peut être affecté à un ou plusieurs rôles, et un rôle peut être attribué à un ou plusieurs utilisateurs. PA est une relation qui permet d'accorder des permissions aux rôles (PA $\subseteq$ PERMS $\times$ ROLES). De même, pour les permissions, une seule permission peut être assignée à plusieurs rôles, et pour un seul rôle, on peut assigner plusieurs permissions. Les permissions accordées aux rôles sont attribuées aux utilisateurs qui acquerront les privilèges du rôle.

SESSIONS représente les sessions de travail des utilisateurs. Une session permet à l'utilisateur, dans sa connexion au système, d'activer un ou plusieurs rôles auxquels il a droit. Chaque session est associée à un seul utilisateur et chaque utilisateur est associé à une ou plusieurs sessions.

- La relation *session_users* spécifie l'utilisateur associé à une session.

  session_users (s : SESSIONS) → USERS

- La relation *session_roles* spécifie les rôles activés par l'utilisateur dans une session

$$\text{session\_roles}(s_i) \subseteq \{r \in \text{ROLES} \mid (\text{session\_users}(s_i), r) \in \text{UA}\}$$

Les permissions accordées à l'utilisateur sont les permissions attribuées aux rôles actifs dans toutes les sessions de l'utilisateur.

1.3.2 Hiérarchie et contraintes de rôles

Dans la figure 1.5, le niveau *Hierarchical RBAC* introduit la relation RH qui ajoute la possibilité de construire une hiérarchie de rôles ($RH \subseteq ROLES \times ROLES$). Cette relation permet de structurer les rôles en vue d'organiser les autorités et responsabilités dans une organisation. RH définit une relation d'héritage par laquelle les rôles supérieurs héritent des permissions des sous-rôles. Dire que le rôle R1 est un rôle supérieur au rôle R2 implique que toutes les permissions accordées à R2 sont aussi accordées à R1. Ainsi, par la relation d'héritage, les sous-rôles obtiennent une adhésion des utilisateurs des rôles supérieurs. Si le rôle R1 est un rôle supérieur au rôle R2, cela implique que tous les utilisateurs affectés au rôle R1 sont aussi affectés au rôle R2. RBAC reconnaît deux types de hiérarchies: *générale* et *limitée*.

- La variante *générale* supporte l'héritage multiple qui permet à un rôle supérieur d'hériter des permissions de deux ou plusieurs sous-rôles ce qui implique l'adhésion de ses utilisateurs aux sous-rôles. La hiérarchie générale définit un ordre partiel sur l'ensemble des rôles, symbolisé par ≥. Elle est traduite formellement par l'expression suivante:

Etant donnés $r1, r2 \in ROLES$

$$r1 \geq r2 \Rightarrow \text{authorized-permissions}(r2) \subseteq \text{authorized-permissions}(r1)$$
$$\wedge\ \text{authorized-users}(r1) \subseteq \text{authorized-users}(r2)$$

avec

$$\text{authorized-users}(r) = \{u \in \text{USERS}, (u,r) \in \text{UA}\}$$
$$\wedge\ \text{authorized-permissions}(r) = \{p \in \text{PRMS}, (p,r) \in \text{PA}\}$$

- La variante *limitée* impose des restrictions sur la hiérarchie de rôles. Un rôle peut avoir un ou plusieurs ascendants, mais il est limité à un seul descendant. Les hiérarchies doivent correspondre à une arborescence simple où un rôle ne peut hériter que d'un seul parent. Ceci est formalisé par la contrainte suivante :

$$\forall\ r, r1, r2 \in \text{ROLES}, r \geq r1 \wedge r \geq r2 \Rightarrow r1 = r2$$

Le niveau *Constrained RBAC* permet la gestion des conflits d'intérêts induits par des rôles incompatibles affectés à un même utilisateur. Pour ce faire, le modèle RBAC propose d'assurer la séparation de devoirs en empêchant l'appartenance d'un utilisateur à des rôles contradictoires ou en exclusion mutuelle. Dans la figure 1.5, deux types de contraintes ont été introduits pour exprimer la séparation de devoirs: les séparations statiques (Static Separation of Duties ou SSoD) et les séparations dynamiques (Dynamic Separation of Duties ou DSoD).

- SSoD empêche l'affectation d'un utilisateur à deux rôles en conflit, et qu'une hiérarchie de rôles amène un utilisateur à obtenir les permissions de deux rôles en conflit.
- Par ailleurs, DSoD impose des restrictions sur les rôles qui peuvent être activés par un utilisateur dans sa connexion au système à travers une session. Elle évite qu'un utilisateur possède deux rôles en conflit dans une même session.

1.3.3 RBAC pour contrôler l'accès au BP

Contrôler l'accès au niveau des workflows d'un BP, selon (Kandala, Sandhu, 2002), « *consiste à assigner aux utilisateurs, conformément aux règles de l'organisation, des permissions pour effectuer certaines tâches au sein de l'organisation en fonction de leurs qualifications et responsabilités* ».

Plusieurs travaux ont proposé des solutions pour adapter le modèle RBAC à la spécification de politiques de contrôle d'accès au niveau des workflows d'un BP. Parmi ces travaux, citons :

(Wainer et al., 2003) définit le modèle W-RBAC, basé sur les rôles, pour contrôler l'accès au niveau des workflows d'un BP. Ce modèle est composé de deux couches : W0-RBAC et W1-RBAC. W0-RBAC combine les éléments du RBAC et du workflow avec une séparation des préoccupations dans l'administration des permissions. Ces permissions sont spécifiées formellement. Cela permet la sélection des utilisateurs autorisés à effectuer des tâches d'un BP. W1-RBAC définit des extensions du modèle W0-RBAC qui intègrent le traitement des exceptions dans l'approche de contrôle d'accès.

(Oh, Park, 2003) propose le modèle Task-Role Based Access Control (T-RBAC) qui adapte le modèle RBAC pour la spécification des politiques d'autorisation en tenant compte des exigences particulières des BPs. T-RBAC propose la classification des différentes tâches qui sont l'unité fondamentale d'une activité d'un BP. Après la classification, il traite chaque tâche différemment selon sa catégorie, et définit un

niveau de contrôle d'accès pour les différentes tâches. T-RBAC supporte également la supervision de la hiérarchie des rôles.

(Botha, Eloff, 2001), (Wolter, Schaad, 2007), (Brucker et al., 2012) et (Strembeck, Mendling, 2011), par exemple, s'intéressent à la spécification des contraintes de la séparation des tâches sur les utilisateurs et les rôles. Ces contraintes visent à éviter les conflits d'intérêt dans l'exécution des activités d'un BP en empêchant des utilisateurs d'exécuter des tâches conflictuelles. Elles considèrent l'ordre et l'historique d'une tâche pour décider si un utilisateur connecté à un ensemble de rôles est autorisé à effectuer certaines tâches.

Les travaux comme par exemple ceux de (Atluri, Warner, 2005), (Gaaloul et al, 2012) et (Schefer, Strembeck, 2011) proposent l'adaptation du modèle RBAC pour supporter la délégation des tâches au niveau du workflow d'un BP. La délégation des tâches implique la délégation d'un ensemble de rôles ou de privilèges d'un utilisateur, à d'autres utilisateurs. Dans (Atluri, Warner, 2005), la délégation est conditionnée par différentes contraintes comme par exemple les contraintes d'autorisation, les contraintes d'activation du rôle, et les exigences de dépendance de tâches.

(Schefer-Wenzl, Strembeck, 2012) intègre les contraintes de contexte dans la spécification de politiques RBAC au niveau de BP. Ces contraintes sont introduites par (Brezillon, Mostefaoui, 2004) face aux nouveaux problèmes de sécurité liés à la mobilité des utilisateurs et à la flexibilité des BPs. (Schefer-Wenzl, Strembeck, 2012) spécifie les contraintes de contexte par des conditions qui comprennent des attributs de contexte et des fonctions de contexte. Un attribut de contexte représente une propriété de l'environnement dont la valeur pourrait être changée dynamiquement (comme par exemple, l'heure, la date ou le lieu). Pour chaque attribut de contexte, une fonction de contexte est définie pour obtenir la valeur actuelle d'un attribut de contexte (par exemple, la fonction *date()* retourne la date du jour).

Dans notre travail nous proposons une approche qui combine les formalismes du modèle RBAC et des diagrammes d'activités d'UML2 pour contrôler l'accès au niveau des activités d'un BP. Notre approche intègre des contraintes contextuelles d'autorisation qui font référence à l'état du système. Ces contraintes permettent de spécifier des politiques RBAC qui dépendent des rôles affectés aux utilisateurs et des aspects dynamiques du système comme par exemple les valeurs des attributs. Ainsi, ce travail propose une formalisation et une validation formelle de la politique RBAC exprimés au travers des diagrammes d'activités de contrôle d'accès.

1.3.4 Profils UML de spécification de politiques RBAC au niveau de BPs

Peu de travaux ont proposé des profils UML pour la spécification de politiques RBAC au niveau de BPs. Les principaux travaux sont UMLsec et l'extension de *Strembeck* & *Mendling*.

1.3.4.1 UMLsec

UMLsec (Jürjens, 2004) propose plusieurs profils pour la spécification des différents besoins de sécurité tels que la confidentialité, l'intégrité, la non répudiation et le contrôle d'accès. Ces profils portent sur les différents diagrammes d'UML comme par exemple le diagramme de classes, le diagramme de composants et le diagramme d'activités.

Dans cette section, nous présentons le profil d'UMLsec qui propose une extension du diagramme d'activités d'UML pour la spécification de politiques RBAC. Ce profil représente un diagramme d'activités dans un package stéréotypé par « rbac ». Il décrit un comportement du système au moyen des actions placées dans des partitions séparées qui précisent les acteurs responsables de leur réalisation.

Le profil *rbac* d'UMLsec utilise trois types de tags : *protected*, *role*, et *right*. Le tag {protected} définit l'ensemble des actions d'un diagramme d'activités qui nécessitent un contrôle d'accès. Le tag {role} est défini par la paire (actor, role). Il représente les acteurs (actor) affectés aux rôles (role) exécutant les actions du diagramme d'activités. Le tag {right} exprime l'affectation des actions protégées aux rôles qui peuvent les utiliser. Il est défini par la paire (role, right) qui signifie que seul l'acteur assigné au rôle défini comme *role* dans la paire peut exécuter l'action définie comme *right* dans la même paire.

Jürjens a illustré son profil par l'exemple de la figure 1.6 qui exprime une politique de contrôle d'accès à une activité de gestion de crédits. L'exemple définit deux acteurs *Employee* et *Supervisor*. L'acteur *Employee* traite les demandes de crédits, inscrit les crédits et fait le transfert de l'argent. L'acteur *Supervisor* autorise les crédits qui dépassent 10000. La politique de contrôle d'accès définie par les tags *role*, *right* et *protected* exprime que seul l'acteur *Supervisor* associé au rôle *credit approver* peut exécuter l'action *authorize credit.*

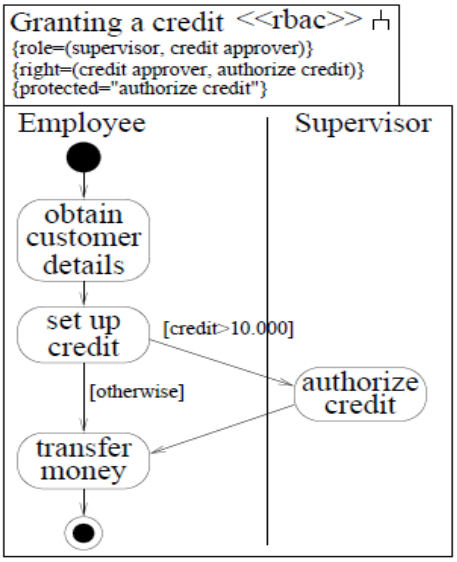

Figure 1.6. Exemple illustrant le profil rbac d'UMLsec

1.3.4.2 Extension de Strembeck & Mendling

(Strembeck, Mendling, 2011) présente une approche basée sur un méta-modèle pour la modélisation des BPs comprenant une spécification RBAC. Une extension du diagramme d'activités d'UML2 a été proposée pour illustrer l'approche. La figure 1.7 présente cette extension qui inclut les éléments du modèle RBAC et du diagramme d'activités. Des nouveaux éléments ont été introduits comme par exemple *BusinessActivity*, *BusinessAction*, *Subject*, *Role*, *RoleToSubjectAssignment*, et *RoleToRoleAssignment*.

Les éléments issus du diagramme d'activités sont *BusinessActivity* et *BusinessAction*. *BusinessActivity* est défini comme une sous-classe de la méta-classe *Activity* du méta-modèle UML. *BusinessAction* spécialise l'élément *Action* du méta-modèle UML en lui associant des propriétés de contrôle d'accès. Ces propriétés définissent le rôle qui peut exécuter l'action et les actions en exclusion (conflit d'intérêts) avec cette action. Le méta-modèle de *Strembeck* & *Mendling* spécifie quatre types de séparation de tâches : *staticExclusion*, *roleBinding*, *subjectBinding* et *dynamicExclusion*.

Les éléments inhérents au modèle RBAC sont : *RoleToSubjectAssignment*, *RoleToRoleAssignment*, *Subject*, et *Role*. Ils permettent de spécifier les rôles et leurs relations hiérarchiques ainsi que la connexion des utilisateurs aux rôles.

Strembeck et *Mendling* ont enrichi la sémantique de leur méta-modèle en associant des invariants à ses éléments.

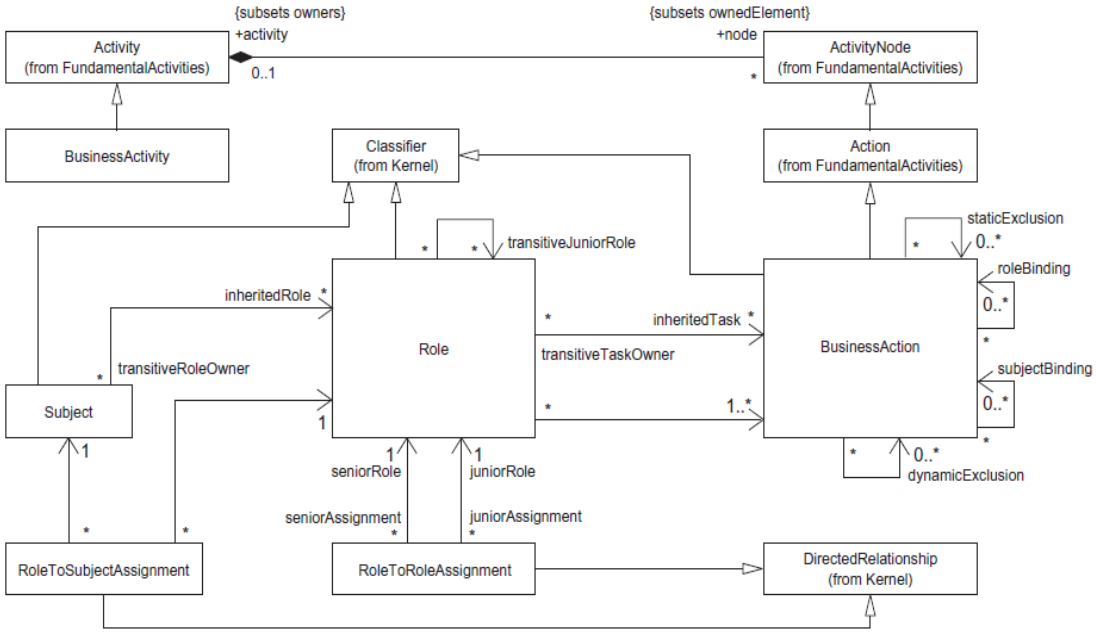

Figure 1.7. Le méta-modèle de Strembeck & Mendling

La figure 1.8 présente un exemple qui illustre le profil de *Strembeck* & *Mendling*. Les éléments de type *BusinessActivity* sont notés par *BA* et les *BusinessAction* par *B*. Les attributs des actions DME (Dynamic Mutual Exclusion), SBind (Subject Binding) et RBind (Role Binding) spécifient les contraintes de séparation de tâches.

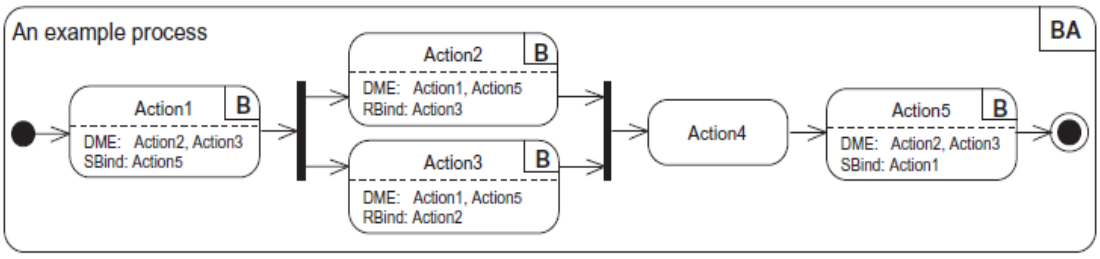

Figure 1.8. Exemple illustrant l'extension de Strembeck & Mendling

1.3.5 Validation formelle de politiques RBAC au niveau de BPs

Plusieurs approches ont été proposées pour la validation formelle de politiques RBAC au niveau des BPs. Citons, par exemple :

(Wolter et al., 2009) propose une approche qui commence par la modélisation des BPs et d'une politique de contrôle d'accès associée à l'aide de la notation BPMN étendue par des éléments de spécification des aspects de sécurité. Cette approche fait ensuite une traduction automatique des modèles dans le métalangage de processus *Promela* dont la sémantique est basée sur CPnets *(Réseaux de Pétri colorés)*. La dernière étape de l'approche utilise le *model checker SPIN* pour vérifier les propriétés de séparation de devoirs (SoD) d'une politique RBAC. Cependant l'approche ne tient pas compte de tous les éléments d'une politique RBAC comme l'affectation de l'utilisateur à de multiples rôles et la hiérarchie de rôles.

(Arsac et al., 2011) propose une approche basée sur le langage de spécification *AVANTSSAR* pour générer un modèle formel de contrôle d'accès au niveau des BPs, et ensuite, il analyse la politique RBAC à l'aide de *model checking*. L'analyse consiste à détecter des attaques et à fournir leurs traces à l'analyste. Cependant l'approche ne supporte pas la représentation graphique de la politique RBAC. En outre, elle ne prend pas en compte des contraintes contextuelles d'autorisation.

Il existe également d'autres travaux qui proposent la formalisation de politiques RBAC associées aux BPs en utilisant d'autres langages tels que *EFSM* (Extended Finite State Machine) (Duty et al., 2007) et *LTL formulae* (Armando, Ponta, 2010).

Notre approche propose la traduction des modèles BAAC@UML en B à l'aide de la plateforme *B4MSecure,* et elle utilise ensuite les outils dédiés à la méthode B pour la validation de la politique RBAC à partir de la spécification formelle. Elle considère des politiques RBAC augmentées avec des contraintes contextuelles d'autorisation dans la formalisation des activités d'un BP. L'activité formelle peut être animée par un utilisateur connecté à des rôles et les contraintes contextuelles sont évaluées au cours de l'animation. Ces contraintes peuvent accorder ou refuser l'accès de l'utilisateur aux différentes tâches de l'activité.

1.4 Conclusion

UML et RBAC sont des techniques de spécifications reconnues en génie logiciel. UML est le langage standard de modélisation des systèmes d'information. RBAC est le modèle standard de contrôle d'accès. L'état de l'art que nous avons effectué ne couvre pas l'intégralité de ces techniques, cependant il donne une idée sur chacune d'entre elles.

Dans le cadre de notre étude, nous spécifions, à l'aide de diagrammes d'activités d'UML 2, et du modèle RBAC, des politiques de contrôle d'accès au niveau des

activités d'un BP. Les chapitres 2 et 3 présentent cette spécification et vérifient sa cohérence.

Nous complétons la spécification graphique en faisant la traduction des diagrammes d'activités de contrôle d'accès en langage B. Les outils dédiés à la méthode B (prouveurs et animateurs) nous permettent de vérifier et de valider la politique de sécurité à partir de la spécification formelle. Le chapitre 4 aborde cette partie.

Le dernier chapitre de cette thèse décrit les outils développés dans le cadre de notre étude.

Chapitre 2

Profil BAAC@UML
(Profil UML de contrôle d'accès aux activités métiers)

2.1 Introduction

Au cours de ces dernières années, le contrôle d'accès aux systèmes d'information est devenu une préoccupation majeure, qui doit être prise en compte dans les phases d'analyse et de conception. Des études récentes proposent quelques profils UML utiles pour la spécification de politiques de contrôle d'accès basées sur le modèle RBAC (voir section 1.3.4). Comme nous avons vu dans l'introduction de cette thèse, SecureUML est un profil qui permet de spécifier une politique RBAC au travers des diagrammes d'UML. Ce profil permet d'exprimer des contraintes contextuelles, appelées aussi contraintes d'autorisation. Ces contraintes portent sur l'état spécifié par le diagramme de classes et conditionnent l'évaluation des permissions. La contrainte d'autorisation est une notion très importante qui permet de relier les concepts fonctionnels et de sécurité.

Dans ce chapitre, nous proposons le profil BAAC@UML (Business Activity Access Control with UML) qui vise à compléter la vue statique de contrôle d'accès par des diagrammes dynamiques. Ce profil étend le diagramme d'activités d'UML pour la spécification des activités d'un BP en tenant compte d'une politique RBAC. BAAC@UML définit des diagrammes d'activités à deux niveaux : le premier est abstrait et permet de décrire une activité métier par une coordination d'actions de haut niveau exécutées par les acteurs du système. Le deuxième est concret et consiste à exprimer les actions abstraites par des actions de bas niveau qui représentent les opérations des classes. Ce diagramme concret tient compte des permissions SecureUML et de leurs éventuelles contraintes d'autorisation qui seraient associées comme préconditions locales aux actions concrètes concernées. Le contrôle d'accès à une activité concrète est aussi réalisé en affectant l'activité à un ou plusieurs rôles. Seuls les utilisateurs assignés à ces rôles peuvent exécuter les actions concrètes de l'activité. La sémantique de nos extensions de diagrammes d'activités est spécifiée au moyen d'un méta-modèle.

Afin de représenter les modèles BAAC@UML et SecureUML, nous avons développé l'outil *Graphical-ACP*. Cet outil utilise l'éditeur graphique de *Papyrus* pour l'édition de modèles de contrôle d'accès des vues statique et dynamique. Le chapitre 5 explique les étapes de sa mise en œuvre.

Dans la section 2.2 de ce chapitre, nous présentons SecureUML, le point de départ de notre approche. La section 2.3 discute la spécification fonctionnelle des activités métiers. La section 2.4 montre comment exprimer le contrôle d'accès aux activités à l'aide de notre profil. La section 2.5 définit le méta-modèle qui permet d'étendre le diagramme d'activités pour la spécification de politiques RBAC. Enfin, la dernière

section conclut le chapitre et compare notre approche avec les travaux similaires qui traitent le contrôle d'accès au niveau des BPs.

2.2 SecureUML

SecureUML (Basin et al., 2006) (Basin et al., 2009) propose un méta-modèle basé sur le modèle RBAC. La figure 2.1 présente ce méta-modèle qui spécifie une syntaxe abstraite définie par les concepts *User, Role, Permission, Action, Resource et AuthorizationConstraint*. Dans ce méta-modèle, les utilisateurs (méta-classe User) sont organisés par rôles (méta-classe Role). Pour chaque accès, à un utilisateur sont assignés des rôles qui lui permettent d'accéder à des permissions (méta-classe Permission) lui autorisant l'exécution d'un ensemble d'actions (méta-classe Action) sur des ressources (méta-classe Resource). Une ressource représente les éléments cibles de sécurité. *Authorization Constraint* est un prédicat logique rattaché à *Permission* pour exprimer les conditions d'accès. SecureUML définit également des relations de généralisation (RoleHierarchy) entre les rôles qui fournissent la possibilité d'héritage des permissions entre eux. *AtomicAction* spécifie une action qui peut être appliquée directement sur *Ressource. CompositeAction* définit des actions abstraites qui peuvent être raffinées en actions atomiques.

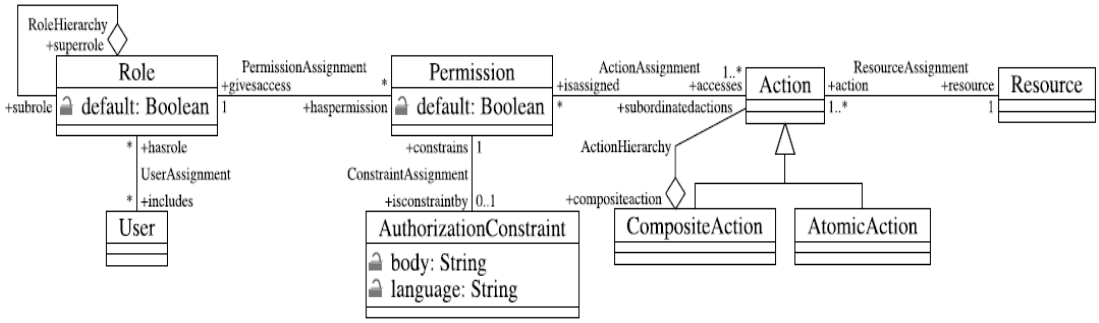

Figure 2.1. Le méta-modèle SecureUML

Le méta-modèle SecureUML est généraliste; il permet d'étendre tout langage de modélisation pour exprimer le contrôle d'accès aux systèmes. Cette extension est réalisée en mettant en relation le méta-modèle SecureUML et le méta-modèle du langage de modélisation à étendre. La relation entre les deux méta-modèles est appelée *Dialecte* et consiste à définir des liens entre leurs méta-classes. Ces liens spécifient les éléments protégés du langage de modélisation et les actions possibles sur eux. Pour le langage UML, deux diagrammes ont été étendus : le diagramme de classes et le diagramme d'états (Brucker, Doser, 2007). Dans le cadre du diagramme de classes, le dialecte spécifie les classes, les attributs, les terminaisons d'associations

et les opérations comme ressources. Le tableau 2.1 présente les actions possibles sur les différentes ressources.

Ressource	Actions
Entité	create, read, update, delete, full access
Attribut	read, update, full access
Opération	execute
Terminaison d'association	read, update, full access

Tableau 2.1. Le dialecte du diagramme de classes

Dans notre approche, nous nous sommes intéressés au diagramme de classes SecureUML qui permet de représenter une politique de sécurité basée sur le modèle RBAC dans une vue statique (voir figure 2.4). Ce diagramme utilise des permissions, représentées par des classes associatives, pour exprimer les règles de contrôle d'accès. Une permission est liée à une classe stéréotypée par «Role» qui représente les utilisateurs affectés au rôle, et une autre classe stéréotypée par «Entity» qui représente la classe cible de la permission. La permission spécifie ainsi les droits d'accès des utilisateurs à l'entité qu'elle protège. Elle est définie par un ensemble d'attributs indiquant les types d'actions autorisées (lecture, modification, etc). Ces attributs sont définis par trois propriétés :

- Le type de la ressource: SecureUML définit quatre stéréotypes pour spécifier le type de la ressource protégée par la permission, « methodaction », « attributeaction », « entityaction » et « associationendaction ». Le stéréotype « methodaction », par exemple, exprime que l'attribut de la permission vise une opération de la classe protégée.
- La ressource: Cette propriété définit le nom de l'attribut de la permission. Elle détermine la ressource à protéger, comme par exemple un attribut, une opération, une terminaison d'association ou une classe.
- L'action: Cette propriété spécifie le type de l'attribut de la permission. Ce type définit l'action autorisée par l'attribut de la permission, comme par exemple, la création, la lecture, la suppression ou la modification.

Il est possible de soumettre une permission à des conditions contextuelles, appelées contraintes d'autorisation. Souvent ces contraintes font le lien entre les informations issues du modèle de sécurité (comme l'utilisateur et ses rôles) et l'état du modèle fonctionnel.

Dans le modèle SecureUML de la figure 2.4, la permission *ParticipantMeeting* contient un seul attribut dont le type de la ressource est *« entityaction »*, la ressource à protéger est la classe *Meeting*, l'action autorisée est *read*. Cette permission est associée à la contrainte d'autorisation *PM-authConstraint.*

2.3 Spécification des activités métiers

Cette section présente notre démarche de modélisation des activités métiers au moyen des diagrammes d'activités. Cela se fait en deux étapes : la première consiste à décrire chaque cas d'utilisation du système par une activité abstraite et la deuxième permet de spécifier une activité abstraite par une activité concrète. Avant de décrire ces étapes, nous présentons l'exemple de planification de réunions qui sera utilisé pour illustrer notre approche.

2.3.1 Exemple d'organisation de réunions

Afin d'illustrer notre travail, nous allons utiliser l'exemple de l'organisation de réunions, proposé initialement par (Feather et al., 1997). Ce système s'adresse à deux types d'utilisateurs : les "initiateurs" planifient des réunions et les "participants" répondent aux invitations des initiateurs.

2.3.1.1 Besoins fonctionnels

Dans un premier temps, nous représentons les différents besoins fonctionnels du système en utilisant le diagramme de cas d'utilisation de la figure 2.2. Ce dernier exprime les différentes façons dont les acteurs peuvent utiliser le système. Les possibilités d'utilisation sont représentées par les *cas d'utilisation* où chaque cas spécifie un comportement attendu du système. Le diagramme de cas d'utilisation représente graphiquement les acteurs reliés par des associations à leurs cas d'utilisation. Chaque association signifie la participation de l'acteur dans la réalisation du cas d'utilisation.

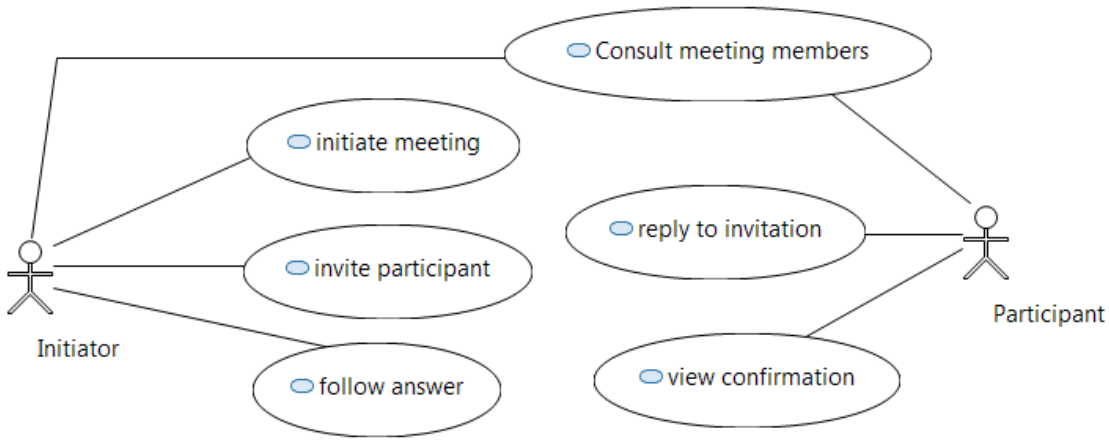

Figure 2.2. Le diagramme des cas d'utilisation du système d'organisation de réunions

Dans la figure 2.2, l'acteur *Initiator* crée des réunions, invite des participants et suit leurs réponses. L'acteur *Participant* répond aux invitations et suit les confirmations des réponses. Les deux acteurs *Initiator* et *Participant* consultent la liste des intervenants de leurs réunions.

Le diagramme de classes d'UML permet de décrire les entités du système (classes et classes associatives), les caractéristiques de chaque entité (attributs de classes), les opérations effectuées sur chaque entité (méthodes de classes) et les relations entre les entités (associations, agrégations, compositions, etc.). La figure 2.3 montre le diagramme de classes du système d'organisation de réunions.

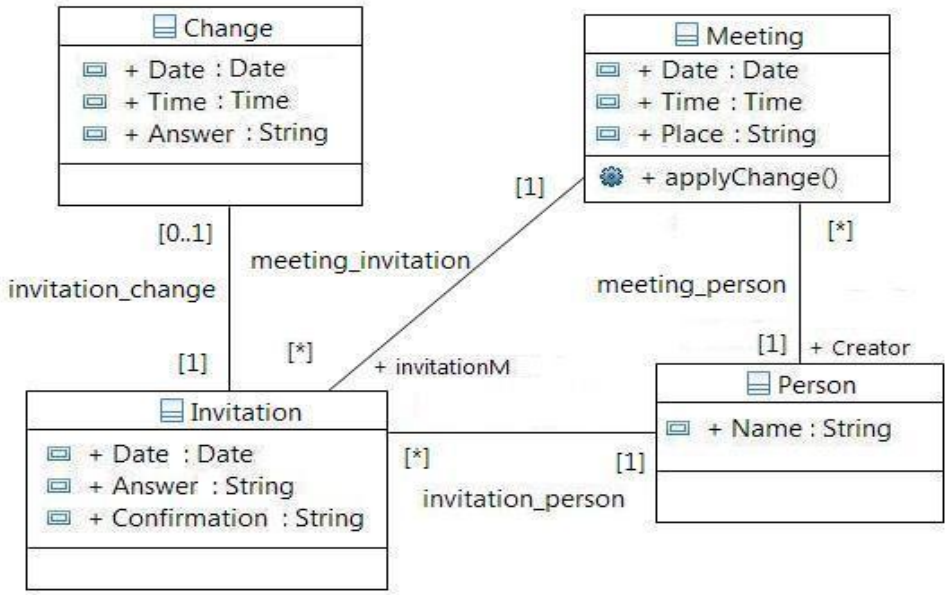

Figure 2.3. Le diagramme de classes du système d'organisation de réunions

Le système permet d'enregistrer les données des personnes (participants et initiateurs), des invitations, des réunions et des propositions de changement, ainsi que les liens entre ces données. Ces liens définissent les personnes créatrices de réunions, les invitations associées aux réunions, les propositions de changement associées aux invitations et les personnes invitées aux réunions.

2.3.1.2 Besoins de sécurité

Afin de protéger les données de l'application, le système d'organisation de réunions applique une politique de contrôle d'accès, basée sur le modèle RBAC, qui répond à deux exigences de sécurité:

- La confidentialité d'une réunion, en assurant que seuls l'initiateur et les participants sont au courant de la réunion.
- L'intégrité d'une réunion, en assurant que seul l'initiateur peut modifier une réunion.

Pour satisfaire ces exigences, le système d'organisation des réunions applique une politique de contrôle d'accès à la classe *Meeting*. Cette classe spécifie les données les plus sensibles qui sont constituées par les données des réunions telles que la date, l'heure et le lieu. La politique de sécurité autorise un initiateur à créer des réunions et à lire et modifier leurs informations. Elle autorise aussi les participants d'une réunion à lire ses informations.

2.3.1.3 Modèle SecureUML

Le modèle SecureUML de la figure 2.4, représenté à l'aide de notre outil *Graphical-ACP*, spécifie le contrôle d'accès à la classe *Meeting*.

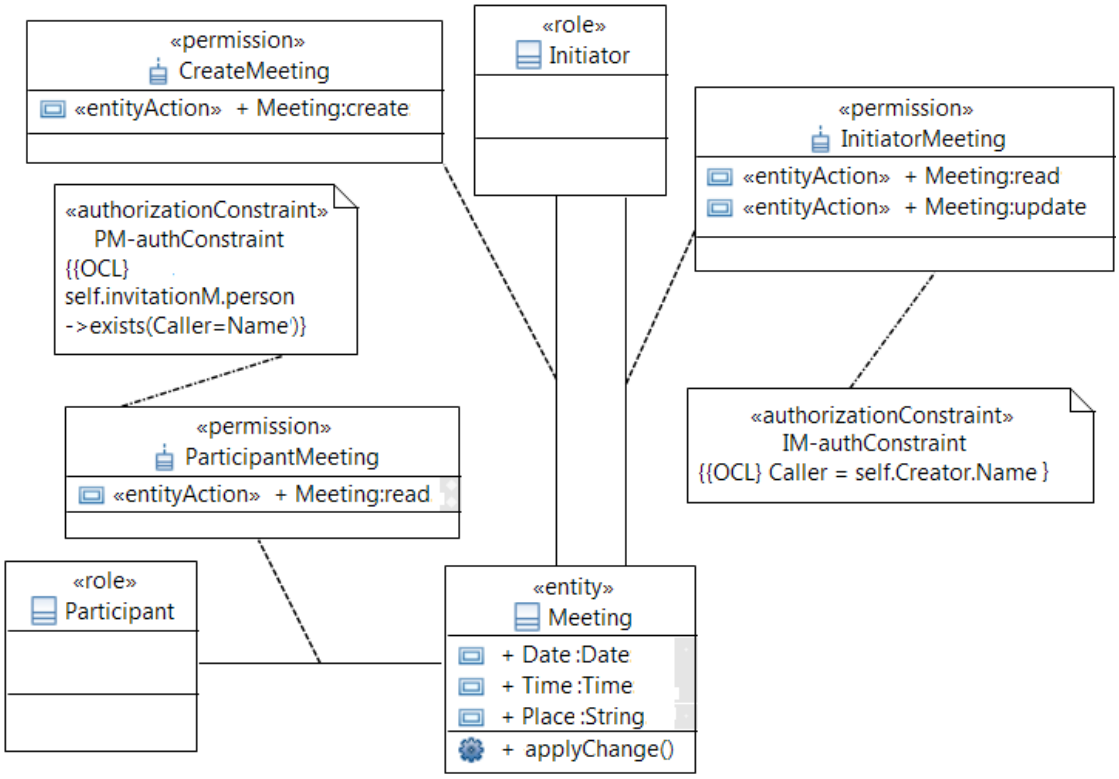

Figure 2.4. Le modèle SecureUML de contrôle d'accès à la classe Meeting

Le contrôle d'accès est assuré par trois permissions :

1. *Create-Meeting* spécifie que seul un initiateur peut créer des réunions.
2. *InitiatorMeeting* exprime qu'une réunion ne peut être lue ou modifiée que par un initiateur, pour autant qu'il soit le créateur de cette réunion. Cette restriction est exprimée par la contrainte d'autorisation associée à la permission.
3. *ParticipantMeeting* exprime que les participants peuvent également lire les informations des réunions, pour autant qu'ils fassent partie des personnes invitées à la réunion.

Les deux permissions *InitiatorMeeting* et *ParticipantMeeting* sont attachées à des contraintes d'autorisation exprimées en OCL (OCL2, 2012). Nous utilisons le mot clé *Caller* du type *String* dans les expressions OCL pour faire référence au nom de l'utilisateur.

2.3.2 Activités abstraites

Dans cette étude, nous utilisons le diagramme d'activités d'UML 2 (UML2, 2011), présenté dans la section 1.2.2.2 (premier chapitre), pour la spécification des activités d'un BP. Plusieurs travaux comme (Russell et al., 2006) et (Sarshar, Loos, 2007) ont traité l'adéquation des diagrammes d'activités d'UML2 pour la modélisation des BPs.

Notre approche de modélisation des BPs commence par la spécification des cas d'utilisation. Plusieurs travaux, tels que (Nurcan et al., 1998) et (Lübke, Schneider, 2008) proposent des approches pour la description des processus métiers sur la base des cas d'utilisation. Un cas d'utilisation représente une famille de scénarios regroupée suivant un critère fonctionnel, chaque cas est une sorte de classe dont l'instance serait le scénario. Dans ce travail, nous utilisons le diagramme d'activités d'UML 2 pour décrire le déroulement d'un cas d'utilisation. Plusieurs auteurs, comme (Roques, 2006), ont spécifié un cas d'utilisation par un ensemble de séquences d'actions qui représentent des tâches exécutées par des acteurs et qui produisent un résultat observable. Les tâches sont des opérations abstraites qui permettent, d'une part, de bien visualiser le comportement d'un cas d'utilisation et d'autre part de communiquer facilement et précisément avec les acteurs du SI. Elles représentent des actions ou des réactions de la part d'un acteur du système pour réaliser un comportement encapsulé dans un cas d'utilisation. Les actions d'un diagramme d'activités peuvent être utilisées pour modéliser les tâches d'un processus métier (Strembeck, Mendling, 2011).

Une activité abstraite permet l'abstraction de la manière dont collaborent des individus pour réaliser un cas d'utilisation. Elle spécifie un cas d'utilisation par un ensemble de tâches en coordination, représentées par des nœuds d'action, pouvant être

exécutées par les acteurs du cas d'utilisation. Les tâches sont l'unité de description de déroulement de l'activité abstraite. Une activité abstraite peut montrer plusieurs scénarios d'exécution. Chaque scénario représente une succession particulière de tâches, s'exécutant du début à la fin de l'activité abstraite.

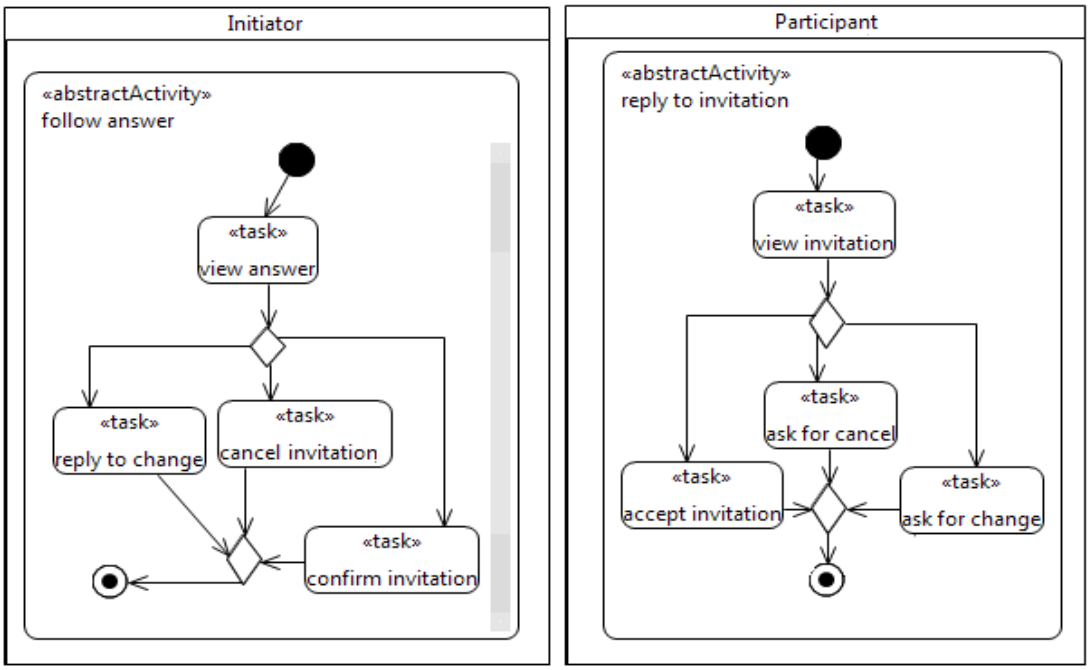

Figure 2.5. Activités abstraites spécifiant les cas d'utilisation reply to invitation et follow answer

Le diagramme d'activités de la figure 2.5, représenté par notre outil *Graphical-ACP*, décrit les deux cas d'utilisation *reply to invitation* et *follow answer* de la figure 2.2 par deux activités abstraites. Les activités sont placées dans des partitions séparées qui précisent les acteurs responsables de ces tâches.

Pour l'activité abstraite *follow answer*, l'initiateur consulte la réponse du participant et suivant cette réponse, il choisit de confirmer l'invitation, d'annuler l'invitation ou de répondre à une demande de changement de la réunion.

2.3.3 Activités concrètes

Les activités concrètes sont construites à partir des activités abstraites. Les activités abstraites décrivent le système comme une boîte noire sans détailler les objets qui le composent. Une fois qu'on dispose d'un diagramme de classes, ces diagrammes abstraits peuvent être raffinés en précisant les objets et les opérations qui les réalisent.

Une activité concrète raffine une activité abstraite en spécifiant le comportement de ses différentes tâches par une coordination d'un ensemble d'actions concrètes qui font référence à des opérations sur les objets des classes. L'exécution de chaque tâche fait appel à une ou plusieurs opérations concrètes. Les opérations sont des fonctions qui peuvent être paramétrées et permettent de manipuler des attributs ou de produire des résultats. Elles représentent un élément de comportement contenu dans une classe et fournissant un service demandé à un objet de la classe. Une opération peut être, par exemple, une affectation de valeurs à des attributs, un accès à la valeur d'une propriété structurelle (attribut ou terminaison d'association), la création d'un nouvel objet ou lien, etc.

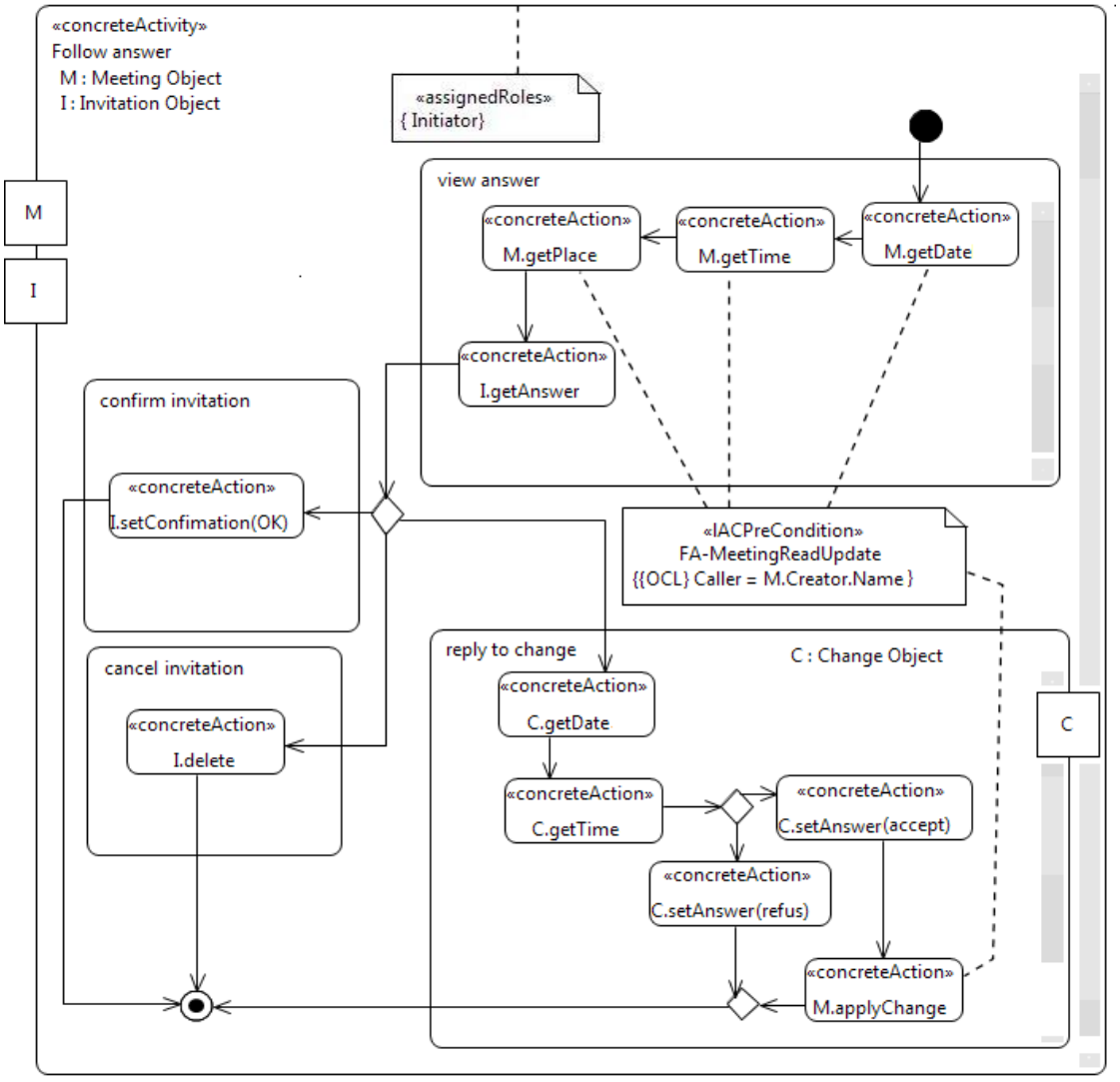

Figure 2.6. Contrôle d'accès à l'activité concrète Follow answer

Le diagramme de la figure 2.6, représenté par notre outil *Graphical-ACP*, raffine l'activité abstraite *follow answer* (la partie gauche de la figure 2.5). Chacune des tâches *view answer* et *reply to change* de l'activité abstraite est décomposée en une coordination d'actions concrètes. La coordination entre les actions et les tâches est représentée par des flots de contrôle et des nœuds de contrôle. Les tâches *confirm invitation* et *cancel invitation* font chacune appel à une seule action concrète.

Les actions concrètes appellent une opération d'une instance de la classe. Ces instances sont définies comme des paramètres associés à l'activité concrète ou à certaines de ses tâches. Ceux-ci fournissent des entrées nécessaires à l'exécution des actions. Dans la figure 2.6, les instances *M* et *I* des classes *Meeting* et *Invitation* sont indispensables pour l'exécution de l'activité *Follow answer*. L'instance *C* de la classe *Change* permet l'exécution de la tâche *reply to change* et reste locale à cette tâche.

Une activité concrète regroupe une famille de scénarios pouvant être exécutés par un utilisateur représentant l'acteur du cas d'utilisation décrit par l'activité. Chaque scénario d'une activité concrète représente une succession particulière d'actions concrètes s'exécutant du début à la fin de l'activité.

2.4 Contrôle d'accès aux activités métiers

La séparation des préoccupations encourage à spécifier indépendamment les aspects fonctionnels et sécuritaires d'un système d'information. Cependant, leur interaction doit être prise en considération au niveau des diagrammes d'activités concrètes. Cette section présente notre approche de spécification d'une politique RBAC au niveau des activités métiers concrètes. Cela se fait à deux niveaux : le premier consiste à affecter l'activité concrète à un ensemble de rôles et le deuxième permet d'associer certaines actions concrètes à des préconditions contextuelles. Dans le deuxième niveau, nous allons montrer comment dans la réalisation d'un BP, les permissions du diagramme SecureUML seront prises en compte. Les permissions et leurs éventuelles contraintes d'autorisation, représentées dans la figure 2.4, sont exprimées par des préconditions dans le diagramme d'activités de la figure 2.6.

2.4.1 Affectation des rôles aux activités

Le diagramme de cas d'utilisation exprime les besoins fonctionnels du système en associant un ensemble de cas d'utilisation à chaque acteur. C'est une sorte de contrôle d'accès qui autorise cet acteur à réaliser les cas d'utilisation qui lui sont associés. Le concept d'acteur introduit dans les diagrammes de cas d'utilisation est très proche du concept de rôle dans le système RBAC. En UML, un acteur est défini comme un rôle

qu'un utilisateur utilise pour interagir avec le SI. Il représente une catégorie d'utilisateurs qui partagent les mêmes fonctions ou activités dans une organisation. Une instance du concept d'acteur est un utilisateur qui interagit avec le système (Goncalves, Hémery, 2000).

Dans notre approche, nous considérons les acteurs associés aux cas d'utilisation comme des rôles. Le diagramme de cas d'utilisation va nous aider donc à trouver l'ensemble des rôles du système. Une activité concrète est affectée à un ou plusieurs rôles. Ce sont les acteurs associés au cas d'utilisation décrit par l'activité. Seuls les utilisateurs assignés à ces rôles peuvent exécuter les tâches et les actions concrètes de l'activité. L'exécution d'une activité concrète correspond à l'exécution d'une succession particulière d'actions concrètes, s'exécutant du début à la fin de l'activité.

L'affectation des rôles aux activités est formalisée au moyen d'une précondition d'activité stéréotypée par «AssignedRoles». La précondition d'activité exprime le fait que toutes les actions doivent être exécutées par le même utilisateur dans un rôle donné. Dans la figure 2.6, celle-ci impose que l'activité soit exécutée par un utilisateur associé au rôle *Initiator*. Ici, elle traduit le fait que le cas d'utilisation *follow answer* est exécuté par l'acteur *Initiator* dans le diagramme des cas d'utilisation (figure 2.2).

2.4.2 Contrôle d'accès aux actions concrètes

Le diagramme de classes SecureUML représente une politique RBAC dans une vue statique. Il spécifie les éléments des classes fonctionnelles qui nécessitent une protection, et il définit les actions qui seront protégées. Cependant, cette représentation ne montre pas à quel moment, dans le BP, cette protection sera invoquée. C'est la mise en place des permissions dans un contexte dynamique qui peut donner une réponse à cette question.

Cette section montre comment exprimer les permissions et leurs contraintes d'autorisation sous forme de préconditions associées aux opérations de classes et comment les propager sur les activités concrètes d'un BP. Cela se fait en deux étapes : la première consiste à sélectionner les opérations critiques parmi les opérations de différentes classes fonctionnelles du système. Après la sélection, la deuxième étape consiste à exprimer les éventuelles contraintes d'autorisation des permissions associées aux opérations critiques comme préconditions des actions concrètes appelant ces opérations.

2.4.2.1 Contrôle d'accès aux opérations

Les attributs des permissions SecureUML représentent des actions abstraites qui donnent lieu à des appels d'opérations concrètes réalisables sur des classes fonctionnelles. Nous considérons comme "critique" toute opération d'une classe protégée qui correspond à un attribut d'une permission. La contrainte d'autorisation associée à la permission exprime une condition obligatoire pour la réalisation des opérations critiques.

La sélection des opérations critiques dépend du type de la ressource concernée par l'attribut de permission. Les règles décrites ci-dessous expliquent comment déterminer les opérations critiques à partir de l'ensemble des opérations de différentes classes fonctionnelles. Ces règles sont définies sous forme de contraintes OCL dans (Basin et al., 2009):

A. Les attributs du type de ressource « methodaction » : seule l'opération définie dans le nom de l'attribut est concernée par la sélection.

B. Les attributs « attributeaction » : la sélection des opérations dépend du type de l'attribut de la permission. S'il s'agit par exemple de la lecture, alors toutes les opérations de la classe cible de la permission qui font une lecture sur l'attribut seront sélectionnées comme opérations critiques. Trois types d'actions sont possibles sur les attributs ayant le type de ressource « attributeaction » : lecture, modification et les deux en même temps (*read, update,* et *full access*) (voir tableau 2.1).

C. Les attributs « associationendaction » : les opérations critiques concernées par ce type diffèrent selon le type de l'attribut, ce sont les opérations qui font soit des lectures, soit des écritures ou soit les deux au même temps (*full access*), sur les terminaisons d'associations définies dans le nom de l'attribut.

D. Les attributs « entityaction » : la sélection pour ce type dépend de la catégorie de l'action (atomique ou composite). Les types d'attribut de permission *read*, *update*, et *full access* fait partie des actions composites, alors que les types *delete* et *create* sont définis comme des actions atomiques. Pour la sélection des opérations critiques, on distingue les cas suivants :

- Pour les types d'attribut *read* et *update*: la sélection concerne toutes les opérations de lecture (le cas de *read*) ou d'écriture (le cas de *update*) sur les attributs et les terminaisons d'associations de la classe protégée.

- Pour le type d'attribut *full access*: l'attribut de la permission est associé aux opérations de création et de suppression de la classe cible de la permission ainsi que toutes les opérations de lecture et d'écriture sur les attributs et les terminaisons d'associations de la classe protégée.

- Pour les types d'attribut *create* et *delete*: dans ces types d'attribut de permission, les opérations de création (le cas de *create*) et de suppression (le cas de *delete*) de la classe protégée sont considérées comme critiques.

Dans le cadre de notre exemple, les opérations autorisées à travers les permissions spécifiées dans le modèle SecureUML de la figure 2.4 se traduisent comme suit :

- Modification de la classe *Meeting* («entityAction» Meeting:update) : permet d'appeler les setters d'attributs *Date, Place* et *Time* de la classe *Meeting*, ainsi que les setters des extrémités d'associations : *Creator, invitationM.* Elle permet également d'invoquer l'opération *applyChange()* qui est une opération de modification.

- Creation de la classe *Meeting* («entityAction» Meeting:create) : permet l'appel au constructeur d'instances de la classe *Meeting*.

- Lecture de la classe *Meeting* («entityAction» Meeting:read) : permet l'appel à toutes les opérations de lecture d'attributs et d'extrémités d'association de la classe *Meeting*.

2.4.2.2 Précondition locale de contrôle d'accès

Certaines actions de l'activité concrète impliquent des opérations critiques dont les permissions sont associées à des contraintes d'autorisation. Il faut tenir compte de ces contraintes d'autorisation dans la spécification de l'activité. Pour ce faire, des conditions supplémentaires, aussi appelées préconditions locales de contrôle d'accès, sont exprimées sous la forme de contraintes stéréotypées comme «LACPreCondition» et reliées aux actions concrètes. Ce sont des contraintes contextuelles exprimées en OCL comme préconditions d'action et permettant l'usage de ces actions en fonction d'informations issues du diagramme de classes. Elles permettent d'empêcher certains utilisateurs assignés aux rôles de l'activité concrète d'exécuter les actions concrètes protégées. En effet, ces contraintes incluent des éléments du modèle RBAC (comme les utilisateurs et leurs rôles) et des éléments du diagramme de classes (comme les attributs et les associations). Elles étendent le modèle RBAC pour exprimer des politiques de contrôle d'accès dépendant des aspects dynamiques du système, tels que les liens entre les instances de classes et l'évolution des valeurs d'attributs.

Les préconditions locales de contrôle d'accès font référence aux contraintes d'autorisation qui portent sur ces actions dans le diagramme d'activités. Dans la figure 2.6, la précondition locale *FA-MeetingReadUpdate* est associée aux quatre actions concrètes *M.getDate, M.getPlace, M.getTime* et *M.applyChange*. Elle garantit que l'utilisateur qui exécute ces actions est le créateur de la réunion: *Caller = M.Creator.Name*, ce qui correspond à la contrainte d'autorisation *IM-authConstraint* de la permission *InitiatorMeeting* dans la figure 2.4.

2.4.3 Optimisation de préconditions

Les préconditions d'activité et les préconditions locales définissent dans quelles conditions les actions concrètes doivent être réalisées. On peut également les voir comme des gardes qui sont évalués lors de l'exécution du diagramme et qui garantissent que la politique de contrôle d'accès est bien respectée. Pour la figure 2.6, on peut se contenter d'évaluer la précondition d'activité en entrée de l'activité, et la précondition locale avant l'exécution de *M.getDate* si les conditions suivantes sont respectées:

1. L'utilisateur n'utilise pas d'autres rôles que celui ou ceux prescrits par la précondition d'activité lors de l'exécution du processus.

2. L'utilisateur ne perd pas le droit d'utiliser ces rôles pendant l'exécution de l'activité.

3. La précondition locale reste vraie pendant l'exécution des quatre opérations concernées. Ce qui signifie que les objets et associations concernées ne sont pas modifiés par les actions du diagramme d'activités ou par des activités extérieures qui seraient menées en parallèles de ce diagramme.

Si ces conditions ne sont pas garanties, il est nécessaire de vérifier les gardes plus souvent et d'encapsuler tout ou partie du diagramme dans une ou plusieurs transactions.

2.5 Extension du méta-modèle de diagrammes d'activités

Le méta-modèle est un mécanisme standard et abstrait préconisé par l'IDM pour définir des modèles. Dans le cadre de l'IDM, tout modèle doit posséder un méta-modèle. L'OMG définit une architecture à quatre niveaux pour répondre à ce principe. Le premier niveau *M0* représente les instances correspond aux objets du monde réel. *M1* est le niveau des modèles qui permet l'abstraction des objets du système en utilisant, par exemple, les différents diagrammes d'UML. *M2* est le niveau du méta-

modèle, il spécifie la sémantique des modèles. Par exemple, l'OMG a défini un méta-modèle pour chaque diagramme UML. *M3* est le niveau de la méta-méta-modélisation. L'OMG utilise le Meta Object Facility (MOF) comme standard pour la définition de méta-modèles.

Le langage UML a été proposé comme une notation générique. De ce fait, les éléments qu'il fournit (par son méta-modèle) ne sont pas prévus pour modéliser des concepts qui sont propres à un domaine (Camus, 2014). Dans notre cas, le diagramme d'activités d'UML 2 peut ne pas représenter un modèle adéquat pour la spécification de politiques RBAC au niveau des activités métiers. Pour répondre à ce besoin de spécialisation, l'OMG propose le mécanisme de profil qui permet de décrire une syntaxe abstraite d'un modèle dédié à un domaine spécifique. Ce mécanisme permet d'étendre des éléments du méta-modèle UML en ajoutant de nouveaux types d'éléments et des contraintes sur les relations entre eux et sur leurs relations avec les autres éléments non spécialisés.

Cette section définit des extensions du méta-modèle du diagramme d'activités d'UML2 pour la spécification de nos modèles BAAC@UML. Ces extensions permettent de spécialiser les éléments du diagramme d'activités pour la modélisation des activités d'un BP en tenant compte d'une politique de contrôle d'accès basée sur le modèle RBAC. Dans la figure 2.10, de nouveaux éléments ont été ajoutés pour spécifier les concepts fonctionnels (activité concrète, activité abstraite, tâche et action concrète) et de sécurité (précondition locale de contrôle d'accès, rôle et utilisateur) inhérents à nos modèles d'activités. Avant de présenter le méta-modèle BAAC@UML, nous présentons les mécanismes de définition des profils UML et des fragments du méta-modèle de diagramme d'activités incluant les éléments utiles pour la spécification de notre profil.

2.5.1 Eléments de spécialisation d'UML

L'OMG définit un ensemble d'éléments pour adapter les diagrammes d'UML à la sémantique d'un domaine particulier. Ces éléments comprennent les stéréotypes, les étiquettes et les contraintes. Le stéréotype est le concept central du profil qui permet de créer les concepts de nouveau domaine. La figure 2.7 présente le fragment du méta-modèle UML décrivant le mécanisme d'extension UML. Cette figure est extraite de la spécification UML (UML2, 2011).

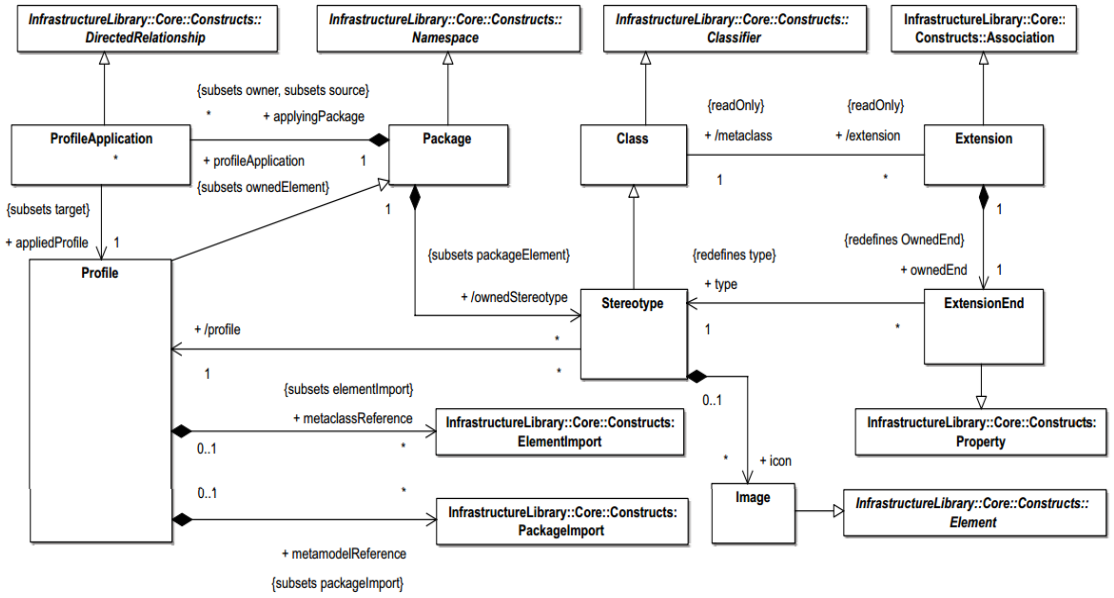

Figure 2.7. Mécanisme d'extension d'UML

Profile est un élément de type *Package* qui permet d'étendre les éléments du méta-modèle d'UML. Il peut contenir des stéréotypes ou des sous-paquetages. Au moyen des éléments *ElementImport et PackageImport*, on peut importer les méta-classes d'UML qui vont être étendues ou des paquetages externes au profil. De ce fait, il est possible d'intégrer d'autres profils ou des concepts définis dans un autre langage de modélisation dans un profil. Un profil doit être appliqué sur un paquetage ce qui permet l'utilisation de ces stéréotypes.

Stereotype est une méta-classe qui doit être créée dans un paquetage profil. Il permet d'étendre tout élément du méta-modèle UML. Cette extension consiste à spécifier un ensemble de propriétés qui caractérisent un concept d'un domaine particulier. Les propriétés sont définies à travers des attributs, des opérations ou des associations avec d'autres éléments du profil. Un stéréotype peut étendre un ou plusieurs éléments d'UML. La relation entre l'élément d'UML étendu et le stéréotype est définie au moyen de l'élément *Extension*, qui est un type d'association. Cette relation permet au stéréotype d'hériter toutes les propriétés de l'élément d'UML étendu. Des images peuvent être associées à un stéréotype pour être utilisées comme icônes à la place ou avec le nom du stéréotype (aucune icône n'est prédéfinie par UML).

Les deux autres éléments de spécialisation d'UML sont les étiquettes et les contraintes. Une étiquette, aussi appelée *tag* ou *valeur marquée*, permet d'ajouter un méta-attribut à un stéréotype. Les valeurs marquées sont définies comme attributs d'un stéréotype. Elles possèdent un nom et un type. Une contrainte est une expression

booléenne qui peut être associée à un ou plusieurs éléments d'un profil dans le but d'enrichir leur sémantique. UML ne spécifie pas une syntaxe particulière pour les contraintes, qui peuvent être exprimées en langage naturel, en pseudocode, par des expressions de navigation ou par des expressions mathématiques. Pour exprimer des contraintes de manière formelle, le langage de contraintes OCL (Object Constraint Language) peut être utilisé.

2.5.2 Fragments du méta-modèle de diagramme d'activités

Les figures 2.8 et 2.9 présentent des fragments du méta-modèle de diagramme d'activités extraits de la spécification UML (UML2, 2011). Ces fragments font apparaître les éléments de modélisation du diagramme d'activités utiles dans notre profil.

Le méta-modèle UML définit *Activity* comme l'élément central du diagramme d'activités. Cet élément spécifie un comportement décrit par un enchainement d'unités dont les éléments simples sont les actions. Dans la section suivante, nous allons étendre l'élément *Activity* pour spécifier les concepts *AbstractActivity* et *ConcreteActivity* de notre profil. Au plus haut niveau d'abstraction, *Activity* est composé essentiellement de deux types d'éléments : *ActivityNode* et *ActivityEdge.*

Dans le fragment de la figure 2.8, *ActivityNode* est une méta-classe abstraite qui représente trois types d'éléments : les nœuds d'action (*Action*), les nœuds d'objet (*ObjectNode*) et les nœuds de contrôle (*ControlNode*). La section 1.2.2.2.1 (premier chapitre) présente plus de détails sur ces éléments.

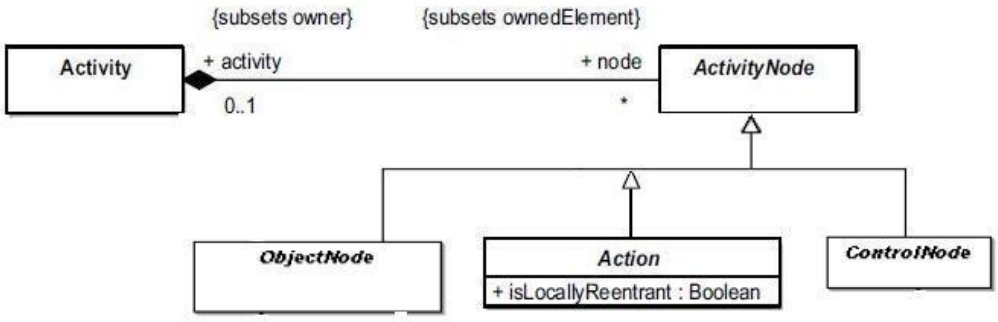

Figure 2.8. Fragment des nœuds d'activité UML

Dans le fragment de la figure 2.9, *ActivityEdge* est une méta-classe abstraite qui représente les flux entre les nœuds d'activités. Deux types de flux peuvent être créés : les flux de contrôle (*ControlFlow)* et les flux d'objets (*ObjectFlow).* La section 1.2.2.2.2 (premier chapitre) présente plus de détails sur ces éléments.

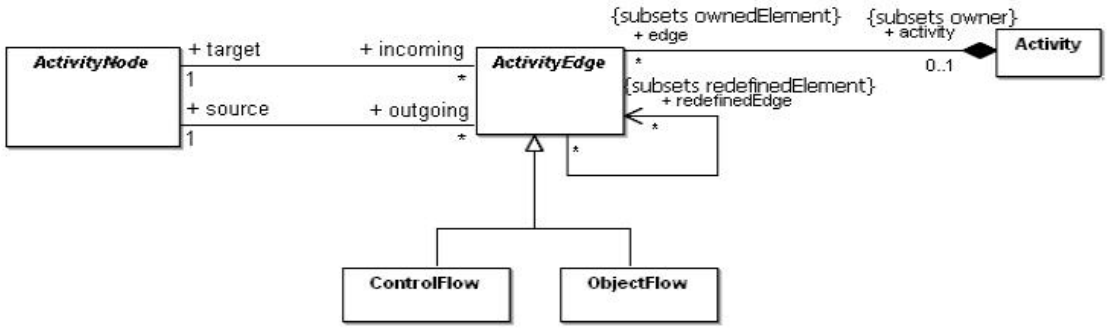

Figure 2.9. Fragment des arcs d'activité UML

2.5.3 Méta-modèle de contrôle d'accès aux activités

La figure 2.10 présente le méta-modèle qui spécifie nos extensions de diagrammes d'activités pour la spécification de politiques RBAC au niveau des activités métiers. Ces extensions sont basées sur le concept de stéréotype qui permet de définir de nouvelles classes d'éléments à partir des éléments du méta-modèle de diagramme d'activités d'UML2. Notre méta-modèle inclut ainsi des éléments du modèle RBAC, tels que les utilisateurs et leurs rôles.

Dans la figure 2.10, les stéréotypes *AbstractActivity* et *ConcreteActivity* étendent l'élément *Activity* du méta-modèle de diagramme d'activités pour spécifier respectivement les activités abstraites et les activités concrètes. L'activité abstraite est réalisée par un ensemble de tâches (stéréotype *Task*) et l'activité concrète est réalisée par un ensemble d'actions (stéréotype *ConcreteAction*).

Les stéréotypes *Task* et *ConcreteAction* sont des extensions de l'élément *Action* du méta-modèle de diagramme d'activités présenté dans la section précédente. Le lien entre le stéréotype *ConcreteAction* et l'élément *Operation* du méta-modèle UML permet d'exprimer l'appel d'opérations fonctionnelles par les actions d'une activité concrète. Nous considérons que ces actions sont contenues dans les tâches réalisant l'activité abstraite. Ce lien de composition permet de garantir une certaine traçabilité entre les activités abstraites et concrètes tout en étant en relation avec les opérations fonctionnelles.

Concernant les concepts de sécurité, le stéréotype *LACPreCondition* étend l'élément *Constraint* d'UML pour exprimer des contraintes contextuelles d'autorisation qui portent sur les actions concrètes. Notre méta-modèle introduit les concepts *User* et *Role* du modèle *RBAC* qui sont représentés par l'élément *Class* du

méta-modèle UML. La classe *User* spécifie l'ensemble d'utilisateurs qui peuvent exécuter les activités métiers. Ces utilisateurs sont organisés en rôles. Une activité concrète est enclenchée par un ou plusieurs utilisateurs assignés à des rôles et est nécessairement issue d'une activité abstraite. L'affectation des rôles aux activités concrètes est formalisée par une précondition d'activité exprimée par une contrainte UML stéréotypée par *AssignedRoles.*

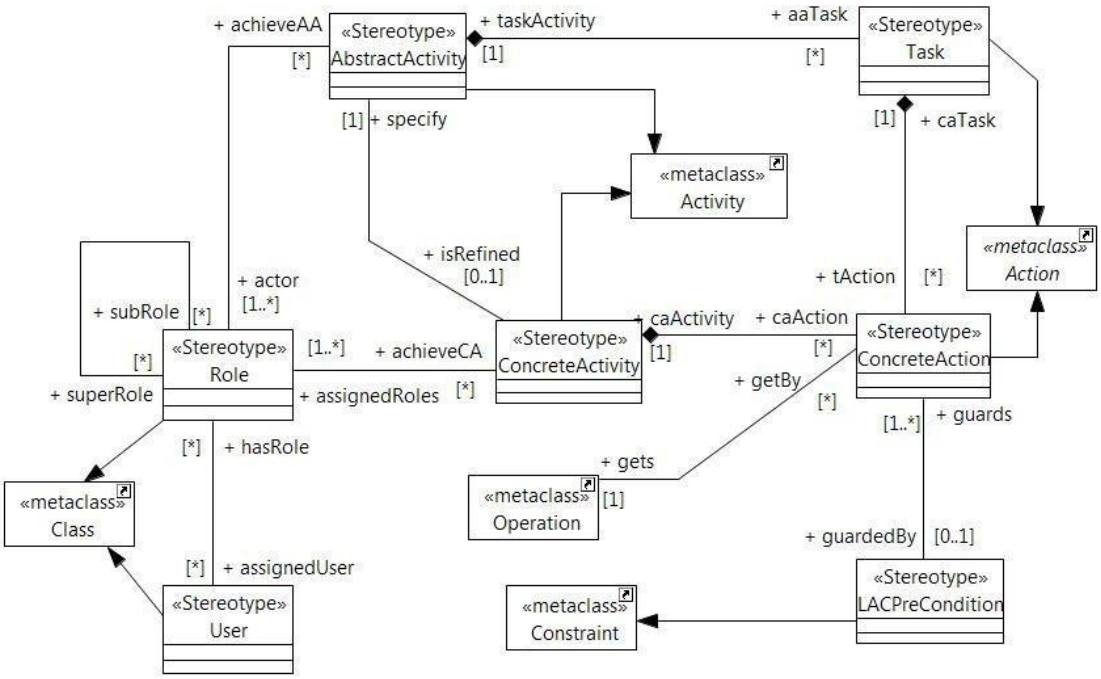

Figure 2.10. Méta-modèle BAAC@UML

Dans notre démarche, les rôles associés à une activité concrète correspondent aux acteurs définis dans l'activité abstraite qu'elle spécifie. Nous établissons l'invariant suivant pour garantir un usage de rôles adéquat durant le processus de spécification allant des activités abstraites aux activités concrètes.

Context ConcreteActivity inv RoleSpecification :
self.assignedRoles → asSet() = self.specify.actor → asSet()

Notre approche considère la relation hiérarchique entre les rôles, par laquelle les rôles supérieurs héritent des permissions des rôles juniors, et les rôles juniors obtiennent une adhésion des utilisateurs des rôles supérieurs. Cela donne aux rôles supérieurs aux rôles affectés à une activité concrète, le droit d'exécuter cette activité. L'invariant *RoleHierarchy*, ci-dessous, assure l'appartenance des super-rôles de

chacun des rôles de l'activité concrète à l'ensemble des rôles affectés à cette même activité ; l'affectation d'un rôle à une activité concrète impute l'affectation de ses super-rôles à la même activité.

Context ConcreteActivity inv RoleHierarchy :
self.assignedRoles → forAll(r:Role|self.assignedRoles → includesAll(r.superRole))

Les deux règles *RoleSpecification* et *RoleHierarchy* sont adjointes à notre méta-modèle pour restreindre l'ensemble des instances valides. Ces règles sont exprimées sous la forme d'invariants de méta-classes et viennent renforcer les règles syntaxiques de notre profil. Dans ces règles, nous avons utilisé le langage OCL qui permet la navigation dans le méta-modèle et dispose de primitives ensemblistes.

Le méta-modèle BAAC@UML de la figure 2.10 a été mis en œuvre dans notre outil *Graphical-ACP*. Cela nous a permis d'utiliser les différents stéréotypes pour la représentation des modèles BAAC@UML des figures 2.5 et 2.6.

2.6 Conclusion

Dans ce chapitre, nous avons présenté le profil BAAC@UML qui propose une extension des diagrammes d'activités d'UML 2 pour garder l'accès des utilisateurs affectés à des rôles aux activités d'un BP. Notre profil introduit la notion de précondition de contrôle d'accès et est basé sur un méta-modèle qui intègre les éléments du méta-modèle du diagramme d'activités et du modèle RBAC. Il permet, outre la spécification de politique RBAC, la prise en compte des contraintes contextuelles d'autorisation qui permettent de formaliser des politiques de contrôle d'accès dépendant des aspects dynamiques du système, tels que l'évolution des valeurs d'attribut. Notre approche considère ainsi la vue statique d'une politique RBAC, exprimée par des modèles SecureUML, dans le contrôle d'accès aux activités.

Notre étude de cas suggère une démarche méthodologique qui débute par la spécification de la vue statique d'une politique RBAC, exprimée au moyen de modèles SecureUML. Ensuite, des diagrammes d'activités abstraites puis concrètes sont réalisés sans tenir compte de la politique de contrôle d'accès. La prise en compte de cette politique se fait alors en deux étapes : la première consiste à repérer les opérations critiques appelées par les actions du diagramme d'activité concrète. Comme nous l'avons vu dans la section 2.4.2, les opérations critiques sont celles correspondant aux actions des permissions SecureUML. La deuxième étape consiste à exprimer les éventuelles contraintes d'autorisation des permissions qui leur sont

associées comme préconditions des actions d'activités concrètes appelant ces opérations critiques.

Notre profil présente des avantages par rapport aux travaux existants de spécification de politiques RBAC en utilisant des diagrammes d'activités, et qui sont présentés dans la section 1.3.4 du premier chapitre. UMLsec ne permet pas de définir des contraintes d'autorisation. De plus, nous n'avons pas connaissance de l'existence d'un méta-modèle qui spécifie la sémantique du profil. L'extension de *Strembeck* et *Mendling* ne prend pas en charge une spécification statique de politique RBAC et des contraintes contextuelles d'autorisation.

Le chapitre suivant présente un ensemble de règles qui vérifient la cohérence entre nos modèles d'activités BAAC@UML et les modèles SecureUML.

Chapitre 3

Cohérence entre les modèles BAAC@UML et SecureUML

3.1 Introduction

Dans le chapitre précédent, nous avons présenté notre profil BAAC@UML qui permet la spécification de la vue dynamique d'une politique RBAC et le profil SecureUML qui permet de modéliser la vue statique d'une politique RBAC. Ces profils offrent aux analystes la possibilité de spécifier des modèles de contrôle d'accès au travers des multiples vues d'UML. Cependant, il est nécessaire de vérifier la cohérence entre leurs modèles. UML se veut être un médium, sous forme de modèle graphique, servant à harmoniser les différents acteurs concourant à la réalisation d'un système, et à garantir la cohérence et la qualité de la conception (Fontan, 2008).

Les incohérences entre les modèles peuvent être une source de failles dans les systèmes (Torre et al., 2014). Plusieurs travaux ont étudié les risques d'incohérences de modèles UML. Selon *Lopez Toro* (Lopez Toro, 2009), la présence d'incohérences dans les modèles UML peut conduire à celle de fautes dans les programmes source déduits et donc de défaillances des programmes exécutables obtenus. Ces défaillances sont d'autant plus dommageables que la fonction du système est critique. (Seuma Vidal, 2006) a identifié quelques fautes révélées par les incohérences, ainsi que les possibles dommages (modèle contenant une faute, transformation erronée aux étapes de raffinement d'un modèle, ou de maintenance d'un même modèle). Pour éviter ces risques, il est nécessaire de détecter et corriger les incohérences au plus tôt.

Dans ce chapitre, nous traitons la question de la cohérence entre les modèles d'activités concrètes de notre profil et les modèles SecureUML. Les deux modèles ne sont pas indépendants ; ils fournissent des informations complémentaires et redondantes et peuvent être en conflit, s'ils sont incohérents. Les spécifications statiques et dynamiques d'une politique RBAC doivent être compatibles, un même élément de modélisation tel qu'un rôle doit toujours avoir la même interprétation; il doit avoir les mêmes droits d'accès dans les deux vues. En effet, les incohérences entre ces modèles peuvent poser beaucoup des problèmes dans leur mise en œuvre.

Selon *Seuma Vidal* (Seuma Vidal, 2006), il existe deux approches de vérification de la cohérence. La première consiste à tester des assertions formulant des propriétés propres aux fonctionnalités du système modélisé (Au niveau M1). La deuxième permet de tester la cohérence des modèles au niveau méta-modèle (Niveau M2). Dans ce chapitre, nous considérons la seconde approche qui définit des propriétés associées aux éléments de méta-modèle UML et qui doivent être respectées par tout modèle UML. Cette approche présente l'avantage d'éviter la reformulation des propriétés, exprimées sous forme de règles de cohérence, pour chaque système.

Ce chapitre propose l'élaboration d'un ensemble de règles de cohérence au moyen du langage OCL (OCL2, 2012). Ces règles vérifient certaines propriétés qui visent à assurer, d'une part, que la spécification SecureUML ne bloque pas la réalisation d'une activité par un utilisateur légitime et, d'autre part, que les activités BAAC@UML respectent les interdictions de la spécification statique SecureUML. En particulier nous nous sommes intéressés à la cohérence inter-modèles qui vérifie la conformité des rôles et des contraintes contextuelles dans les modèles de différentes vues. Dans cette étude, nous avons développé l'outil *ACP-Consistency* mettant en œuvre les différentes règles de cohérence. Cet outil permet de détecter la violation des propriétés invariantes dans les modèles BAAC@UML et SecureUML.

La section 3.2 de ce chapitre présente les différents types de cohérences de modèles UML et les techniques utilisées pour leur vérification. La section 3.3 définit le méta-modèle qui fait le lien entre les concepts de profils BAAC@UML et SecureUML. La section 3.4 présente les règles qui assurent la cohérence entre les diagrammes de deux vues. Ces règles seront implémentées dans la section 3.5 et vérifiées au moyen d'un ensemble de cas de test. Dans la section 3.6, nous mettons au point le système d'organisation de réunions en corrigeant ses différents modèles de contrôle d'accès à l'aide de notre outil *ACP-Consistency*. La section 3.7 conclut le chapitre.

L'étude de la cohérence qui sera présentée dans ce chapitre a été publiée dans la revue ISI (http://isi.revuesonline.com/article.jsp?articleId=36419). Les illustrations sont basées sur le modèle BAAC@UML d'activités concrètes de la figure 2.6 et le modèle *SecureUML* de contrôle d'accès à la classe *Meeting* (figure 2.4), qui ont été présentés dans le chapitre précédent.

3.2 Cohérence des modèles UML

La modélisation de certains systèmes à l'aide d'UML est constituée de centaines de diagrammes développés par des dizaines d'ingénieurs. La détection des incohérences dans ces diagrammes est un enjeu scientifique et industriel majeur (Malgouyres, 2007). Nous présentons ici une classification des différents types de cohérence qui peuvent être considérés dans les modèles UML et les différentes approches vérifiant ces types. Nous positionnons ainsi notre étude dans le cadre de ces approches.

3.2.1 Types de cohérence

(Elaasar, Briand, 2004) présente les classifications, de différents types de cohérence dans les modèles UML, proposées dans la littérature. Il définit cinq classifications :

a) *Syntaxique et Sémantique* : cette classification (Engels et al, 2001) est basée sur le type des propriétés qui doivent être respectées. La cohérence syntaxique assure les règles de *bonne formation* (well-formedness rule) qui vérifient la conformité structurelle d'un modèle par rapport à la syntaxe abstraite spécifiée dans le méta-modèle. La cohérence sémantique définit des règles qui expriment le sens donné aux constructions du méta-modèle. Dans notre étude, nous spécifions des règles qui définissent la sémantique des liens entre les éléments de notre méta-modèle et du méta-modèle SecureUML.

b) *Statique et Dynamique* : la classification de (Sourrouille, Caplat, 2003) considère le moment de la vérification des règles. La cohérence dynamique représente des propriétés d'un modèle qui seront vérifiées lors de son exécution. La cohérence statique définit des règles qui doivent être vérifiées sans l'exécution du modèle. Nous nous intéressons aussi bien à la cohérence statique dans ce chapitre. Dans le chapitre suivant, nous traitons la cohérence dynamique.

c) *Intra-modèle et Inter-modèles* : selon la classification de (Kuzniarz, Staron, 2003) et de (Huzar et al., 2005), la cohérence est soit intra-modèle (appelée aussi horizontale), i.e. une propriété d'un modèle affirmant sa conformité syntaxique et sémantique, soit inter-modèles (également appelée verticale), entre deux modèles en relation mais de types différents. Notre approche permet d'assurer la cohérence verticale entre les modèles d'activités concrètes BAAC@UML et les modèles SecureUML.

d) *Multi-niveaux* : selon (Sourrouille, Caplat, 2002), la cohérence peut être considérée à cinq niveaux différents :

- Le premier niveau définit des règles liées à la syntaxe et à la sémantique des éléments du méta-modèle UML.
- Le deuxième niveau porte sur les extensions du méta-modèle UML (Profils).
- Le troisième niveau spécifie des règles liées au processus de développement.
- Le quatrième niveau définit des règles liées à des contraintes de mise en œuvre.
- Le dernier niveau porte sur le domaine qui représente le système réel spécifié par le modèle.

Les règles de cohérence définies dans la section 3.4 sont positionnées dans le deuxième niveau selon la classification de *Sourrouille et Caplat.*

e) *Nature de l'erreur* : cette classification regroupe les règles suivant la nature de l'erreur. On distingue, par exemple, les règles qui empêchent les contradictions et les règles qui évitent les incomplétudes dans les modèles (Lange et al., 2003). Dans la section 3.4, nous présentons des règles qui empêchent toute contradiction entre les modèles BAAC@UML et SecureUML.

3.2.2 Approches de vérification de la cohérence

(Allaki et al., 2015) présente les approches existantes qui permettent de vérifier les cohérences dans les modèles UML. Nous décrivons ici quelques approches.

Certaines de ces approches utilisent des techniques formelles comme l'algèbre de processus ou la logique. Ces techniques consistent à transformer les modèles UML et leurs règles de cohérence dans un langage formel et ensuite à détecter les incohérences à l'aide de mécanismes d'inférence du langage formel.

Parmi les travaux utilisant les méthodes formelles : (Amalio *et al.*, 2004) transforme les modèles UML en langage Z pour la vérification de leur cohérence. (Laleau, Polack, 2008) utilise le langage B pour la formalisation des modèles UML et leurs règles de cohérence. Dans le chapitre suivant, nous allons également utiliser le langage B pour la formalisation de nos modèles d'activités BAAC@UML.

Parmi les travaux basés sur la logique : (Gallardo et al., 2002) utilise la logique temporelle pour vérifier la cohérence entre un diagramme états-transitions et un diagramme de séquence. (Straeten et al., 2007) vérifie la cohérence des modèles UML au moyen de la logique descriptive.

Pour les travaux basés sur l'algèbre de processus, on peut citer (Rasch, Wehrheim, 2003) qui assure la cohérence entre le diagramme de classes et les diagrammes états-transitions, (Engels et al., 2002a) et (Engels et al. 2002b) qui utilisent les processus CSP (Communication Sequential Process) et (Lam, Padget, 2005) basé sur le *pi-calculus*.

Un autre groupe d'approches propose de contrôler les documents XML issus de modèles UML pour vérifier la cohérence. L'architecture *xlinkit* de ces documents, qui intègre un langage basé sur la logique du premier ordre adapté à des documents XML, permet de faire cette vérification (Bazex et al., 2003). (Gryce et al., 2002) et (Chen, Motet, 2009), par exemple, traitent la cohérence sur la base des documents XML.

D'autres approches comme (Gogolla et al., 2009) et (Chiorean et al., 2004) proposent l'utilisation du langage OCL. Il s'agit ici d'exprimer les règles de cohérence directement sur le méta-modèle d'UML. Ces approches sont basées sur un langage de contraintes riche, intégré dans UML et supporté par plusieurs outils.

Dans ce travail, notre choix a porté sur cette dernière approche et la première approche qui utilise les méthodes formelles. Ce chapitre définit un ensemble de règles exprimées en OCL pour vérifier la cohérence structurelle entre les diagrammes d'activités concrètes de notre profil et les modèles SecureUML. Dans le chapitre

suivant, nous proposons une approche de transformation de modèles BAAC@UML et SecureUML en langage B dans le but de vérifier la cohérence sémantique entre eux.

Avant de présenter les règles OCL, nous spécifions les liens entre les concepts de deux profils.

3.3 Liens entre les méta-modèles BAAC@UML et SecureUML

La cohérence inter-modèles est définie à partir de la relation entre les modèles (Sourrouille, Caplat, 2003). Notre étude de la cohérence entre les modèles BAAC@UML et SecureUML commence par la spécification des liens entre eux. Pour ce faire, nous définissons la sémantique des liens entre ces modèles au moyen de leurs méta-modèles respectifs. L'explicitation de ces liens permet de définir les contraintes qui garantissent la cohérence entre leurs modèles.

Le diagramme de la figure 3.1 montre les méta-modèles SecureUML et BAAC@UML avec leurs liens. Trois liens sont définis entre les deux méta-modèles: à travers le concept du rôle (Role) qui fait partie des méta-modèles de deux profils, à travers l'opération de classe (méta-classe Operation) qui est la cible de sécurité et enfin à travers le lien de traçabilité entre les concepts *LACPreCondition* du profil BAAC@UML et *AuthorizationConstraint* du profil SecureUML.

En SecureUML, les opérations critiques (criticalOperation) sont associées à des actions abstraites qui font partie des permissions. Ces dernières définissent les rôles autorisés à exécuter ces opérations et éventuellement les contraintes d'autorisation pré-conditionnant leur exécution.

En BAAC@UML, Les actions d'une activité concrète (ConcreteAction) font appel à deux types d'opérations : les opérations critiques auxquelles une ou plusieurs permissions sont associées dans le modèle SecureUML (comme par exemple l'opération *getDate()* de la classe *Meeting*) et les opérations non-critiques qui ne sont pas associées à des permissions.

Pour les opérations critiques, nous projetons les éventuelles contraintes d'autorisation qui leur sont associées, sous forme de préconditions des actions concrètes. De ce fait, l'appel d'une opération de classe dans une activité concrète devient préconditionné par la satisfaction de contraintes d'autorisation.

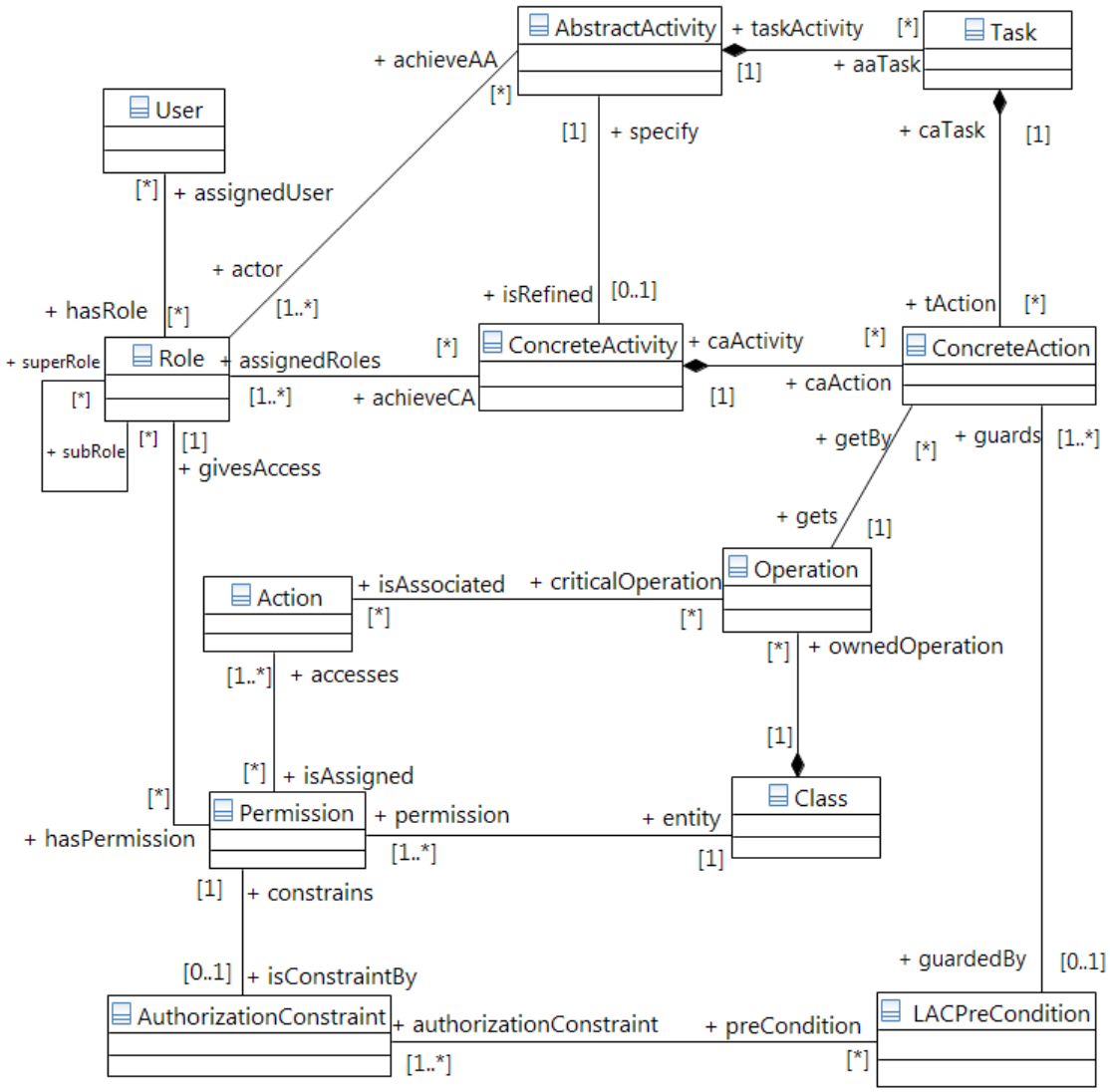

Figure 3.1. Liens entre les méta-modèles BAAC@UML et SecureUML

Après avoir déterminé les liens entre les méta-modèles BAAC@UML et SecureUML, nous spécifions les règles qui assurent la cohérence entre leurs modèles.

3.4 Règles de cohérence

Dans notre étude, nous considérons la cohérence verticale entre les modèles BAAC@UML et les modèles SecureUML. Les mêmes éléments comme par exemple des rôles ou des contraintes contextuelles (*LACPreCondition* et *AuthorizationConstraint*) peuvent apparaître dans les deux modèles. Cela peut introduire des contradictions. Par exemple, la même opération fonctionnelle peut être bloquée par le modèle SecureUML et autorisée par les modèles d'activités concrètes BAAC@UML. Pour éviter ces contradictions, nous spécifions au moyen

d'expressions OCL les règles de cohérence que les instances de notre méta-modèle doivent respecter. Ces règles OCL seront adjointes au méta-modèle pour restreindre l'ensemble des instances valides.

La cohérence des rôles assure que tous les rôles associés à une activité concrète ont, dans le diagramme SecureUML, les permissions d'exécuter toutes les actions de l'activité. La cohérence des contraintes contextuelles assure que les préconditions locales de contrôle d'accès associées à une action concrète font référence aux contraintes d'autorisation adéquates dans les modèles SecureUML. Dans la section 3.4.1, nous définissons un ensemble de requêtes qui seront utilisées pour exprimer les règles de cohérence. Ensuite, nous introduisons les règles de cohérence dans les sections 3.4.2 et 3.4.3.

3.4.1 Requêtes OCL

a) isCritical() indique si une opération donnée est critique. Une opération critique est une opération de classe associée à des permissions SecureUML.

Context** Operation :: isCritical( ) : **Boolean
***body**: self.isAssociated → notEmpty( )*

b) criticalOps() permet de récupérer l'ensemble des opérations critiques pour une activité concrète. Par exemple, les opérations critiques appelées par les actions de l'activité *Follow answer* de la figure 2.6 sont : *getDate(), getPlace(), getTime() et applyChange()* de la classe *Meeting*. Les autres opérations ne sont pas associées à des permissions dans le modèle SecureUML de la figure 2.4.

Context** ConcreteActivity :: criticalOps( ) : **Set(Operation)
***body**: self.caAction.gets → select(o:Operation | o.isCritical( ))*

c) isGuardedAction() teste si une action concrète est associée à une précondition locale de contrôle d'accès. Par exemple : les actions gardées de l'activité *Follow answer* de la figure 2.6 sont : *M.getDate, M.getPlace, M.getTime et M.applyChange*.

Context** ConcreteAction :: isGuardedAction( ) : **Boolean
***body**: not self.guardedBy → isEmpty( )*

d) *OAPermission()* permet de récupérer les permissions SecureUML associées à une action concrète. Ce sont les permissions associées à l'opération critique appelée par cette action et qui sont affectées aux rôles de l'activité concrète incluant l'action.

Chacune des quatre actions *M.getDate, M.getPlace, M.getTime et M.applyChange*, par exemple, est associée à la permission *InitiatorMeeting*.

Context** ConcreteAction :: OAPermission() : **Set(Permission)
***Body** : self.caActivity.assignedRoles.hasPermission → asSet( )*
→ intersection(self.gets.isAssociated.isAssigned → asSet( ))

3.4.2 Conformité des rôles

Une activité concrète est associée à un ou plusieurs rôles. Les utilisateurs assignés à un de ces rôles sont autorisés à exécuter l'activité. L'invariant *ActivityMissingPermissionToLegitimateUser* ci-dessous permet de garantir que tous les rôles autorisés à déclencher une activité concrète disposent de permissions dans le modèle SecureUML leur donnant le droit de réaliser toutes les opérations critiques de l'activité. Dans la section 3.6.1, nous présentons un exemple de violation de cet invariant. Le rôle *Initiator* affecté à l'activité concrète *Follow answer* de la figure 2.6 n'est pas autorisé à exécuter toutes les opérations critiques associées aux permissions dans le modèle SecureUML de la figure 3.9. Par exemple, ce rôle n'a pas la permission d'exécuter l'opération *getAnswer()*.

Context** ConcreteActivity **inv
ActivityMissingPermissionToLegitimateUser :
self.assignedRoles →
***forAll**(r : Role | r.hasPermission.accesses.criticalOperation*
*→ **includesAll**(self.criticalOps( )))*

Il faut noter que cet invariant impose que le plus faible rôle associé à l'activité ait les droits sur toutes les permissions. Dans la suite de nos travaux, nous étudierons d'une part la possibilité de relâcher cette contrainte en s'assurant que l'utilisateur ait des droits pour chaque opération, mais en lui permettant d'utiliser des rôles différents et complémentaires. Il serait également intéressant d'exprimer explicitement le changement de rôle d'un utilisateur lors de l'exécution d'une activité concrète.

Cependant, l'invariant précédent a été défini dans le contexte d'une activité concrète. S'il n'est pas vérifié, il identifiera l'activité erronée mais pas l'opération qui pose problème. Dès lors, nous introduisons une nouvelle contrainte, dans le contexte d'une action concrète qui nous permettra d'identifier les actions non permises au rôle. L'invariant *ConcreteActionMissingPermissionToLegitimateUser* permet d'assurer la conformité des rôles au niveau de chaque action concrète. Il permet d'assurer que

l'ensemble des rôles autorisés à exécuter une action concrète est inclus dans l'ensemble des rôles autorisés par des permissions SecureUML à exécuter l'opération critique appelée par cette action. La section 3.6.1 montre un cas de violation de cet invariant. L'action *I.getAnswer* de l'activité concrète *Follow answer* dans la figure 2.6 ne dispose pas d'une permission dans le modèle SecureUML de la figure 3.9 pour être exécutée par un utilisateur assigné au rôle *Initiator*.

Context** ConcreteAction **inv
ConcreteActionMissingPermissionToLegitimateUser:
*self.gets.isCritical( ) **implies***
self.gets.isAssociated.isAssigned.givesAccess
*→ **includesAll**(self.caActivity.assignedRoles)*

3.4.3 Conformité des contraintes contextuelles

Une action concrète gardée est associée à une ou plusieurs permissions SecureUML. Il faut s'assurer que les contraintes d'autorisation associées à ces permissions sont reliées à la précondition de contrôle d'accès locale gardant cette action d'activité. Cela permet aux rôles associés à l'activité d'exécuter cette action. Nous avons défini trois invariants qui permettent de souligner progressivement tous les types d'incohérences au niveau des contraintes contextuelles. L'invariant *MissingPermissionAuthorizationConstraint* ci-dessous permet de garantir que toute action gardée d'une activité concrète est associée à au moins une permission rattachée à une contrainte d'autorisation et associée à un des rôles de l'activité. Par exemple, dans la figure 2.6, si l'action *C.getDate* était rattachée à la précondition locale *FA-MeetingReadUpdate*, cet invariant serait violé, car l'action n'est associée à aucune permission et donc à aucune contrainte d'autorisation.

Context** ConcreteAction **inv
MissingPermissionAuthorizationConstraint:
*self.isGuardedAction( ) **implies** self.OAPermission( )*
*→**exists**(P : Permission | P.isConstraintB y→ notEmpty( ))*

L'invariant *NotConformanceOfAuthorizationConstraint* ci-dessous permet de garantir que la précondition de contrôle d'accès locale gardant une action concrète fait référence à toutes les contraintes d'autorisation des permissions associées à cette action, et à aucune autre contrainte. Par exemple, cette contrainte serait violée si l'action *I.setConfirmation(OK)* de la figure 2.6 était rattachée à *FA-MeetingReadUpdate*, qui ne correspond pas à la contrainte d'autorisation *II-authConstraint* de la figure 3.9.

Context** ConcreteAction **inv
NotConformanceOfAuthorizationConstraint :
*self.isGuardedAction() **implies***
self.OAPermission().isConstraintBy → asSet()
= self.guardedBy.authorizationConstraint → asSet()

La contrainte suivante s'applique aux actions non gardées. L'invariant *MissingConcreteActionLACPC* assure que les contraintes d'autorisation associées aux permissions sont bien propagées sur les préconditions locales rattachées aux actions concrètes concernées. Dans ce cas, toute action concrète non gardée ne doit pas être associée à des permissions rattachées à une contrainte d'autorisation. Un exemple de violation de cet invariant est présenté dans la section 3.6.1. Dans la figure 2.6, l'action *I.setConfirmation(OK)* de l'activité concrète *Follow answer* n'est pas gardée par une précondition locale. Cependant, cette action est associée à la permission *InitiatorInvit* qui est rattachée à la contrainte d'autorisation *II-authConstraint* dans le modèle SecureUML de la figure 3.9.

***Context** ConcreteAction **inv** MissingConcreteActionLACPC:*
not self.isGuardedAction() and self.gets.isCritical()
***implies** self.OAPermission().isConstraintBy → isEmpty()*

3.5 Validation des règles de cohérence

Pour valider nos règles de cohérence, nous avons développé l'outil *ACP-Consistency* en utilisant l'environnement *EcoreTools* de la plateforme *EMF*[7] (Eclipse Modeling Framework) d'Eclipse. La section 5.3 (chapitre 5) explique la mise en œuvre de l'outil qui spécifie le méta-modèle de la figure 3.1 ainsi que les différents opérations et invariants OCL présentés précédemment. Cet outil permet la création des instances à partir des modèles BAAC@UML et des modèles SecureUML. L'évaluation des différentes règles de cohérence est réalisée à partir de ces instances. Suivant les conventions de (Legeard et al., 2002), ces instances constituent des cas de test positifs et négatifs.

Cette section présente les activités de validation des règles de cohérence à l'aide de notre outil *ACP-Consistency*. Elles consistent à créer des instances à partir des modèles de contrôle d'accès statiques et dynamiques, à interroger ces instances pour vérifier les requêtes OCL et à définir des cas de test pour vérifier les différentes règles de cohérence.

[7] *https://eclipse.org/modeling/emf/*

3.5.1 Création et interrogation des instances

A l'aide de l'outil *ACP-Consistency*, nous avons défini manuellement une centaine d'instances de notre méta-modèle de la figure 3.1, sur lesquelles nous avons évalué les contraintes. La figure 3.2 présente des instances en format XMI. Ces instances correspondent au modèle d'activité concrète *Follow answer* de la figure 2.6 et au modèle SecureUML de la figure 2.4 qui sont présentés dans le chapitre 2. La figure montre l'instance *Follow answer* de la méta-classe *ConcreteActivity* reliée à d'autres instances de méta-modèle BAAC@UML et de méta-modèle SecureUML.

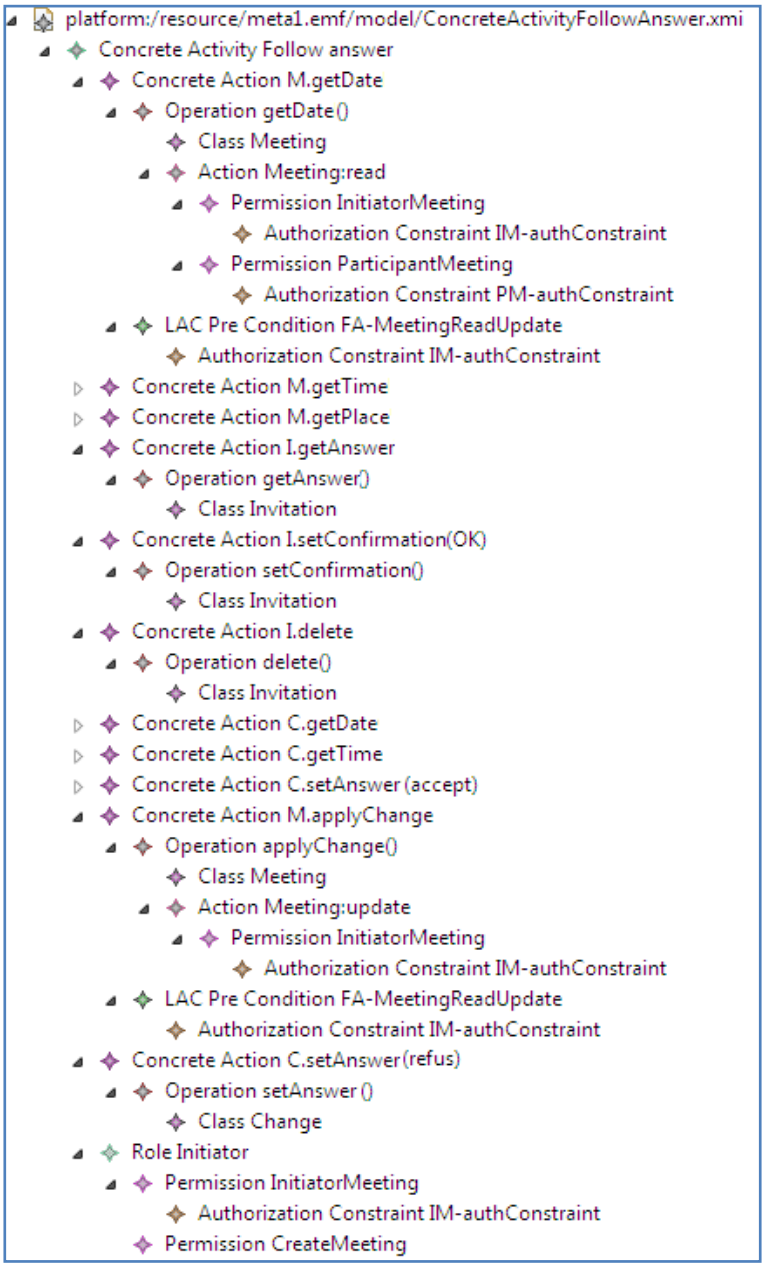

Figure 3.2. Exemple d'instances

Le rôle *Initiator* affecté à l'activité concrète *Follow answer* est associé aux deux permissions *CreateMeeting* et *InitiatorMeeting*. Les trois actions concrètes *M.getDate*, *M.getPlace* et *M.getTime* font appel à des opérations de lecture qui sont associées à l'action *Meeting:read* de permissions *InitiatorMeeting* et *ParticipantMeeting*. L'action concrète *M.applyChange* fait appel à une opération associée à l'action *Meeting:Update* de la permission *InitiatorMeeting*.

La précondition locale de contrôle d'accès *FA-MeetingReadUpdate* se réfère à la contrainte d'autorisation *IM-authConstraint* associée à la permission *InitiatorMeeting*.

L'outil *Interactive OCL* que nous avons intégré dans notre application *ACP-Consistency* permet d'interroger les instances correspondant aux modèles BAAC@UML et SecureUML. Cette interrogation permet de tester les requêtes OCL de la section 3.4.1.

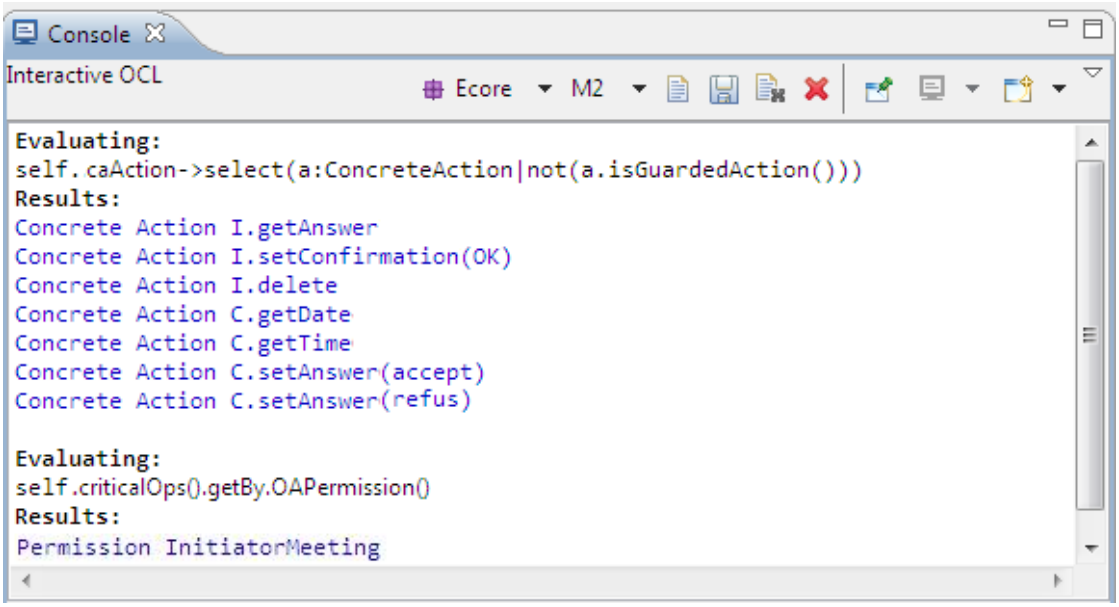

Figure 3.3. Exemple d'interrogation d'instances

Les deux requêtes de la figure 3.3 sont définies dans le contexte de l'instance *Follow answer* de la figure 3.2. La première affiche les actions concrètes non gardées de l'activité et la deuxième affiche les permissions associées aux actions qui font appel aux opérations critiques de l'activité. Ces requêtes permettent de tester les opérations OCL *criticalOps(), isGuardedAction()* et *OAPermission()* présentées dans la section 3.4.1.

3.5.2 Tests positifs

Les tests positifs portent sur des instances cohérentes. Ils sont destinés à se terminer par un succès. Cela signifie qu'aucune de ces instances ne viole les propriétés invariantes spécifiées par les règles de cohérence. Le modèle SecureUML de la figure 2.4 et le modèle d'activités concrètes *Follow answer* de la figure 2.6 présentent un exemple de cas de test positif. Le message de la figure 3.4 est affiché après la validation des modèles par notre outil *ACP-Consistency*.

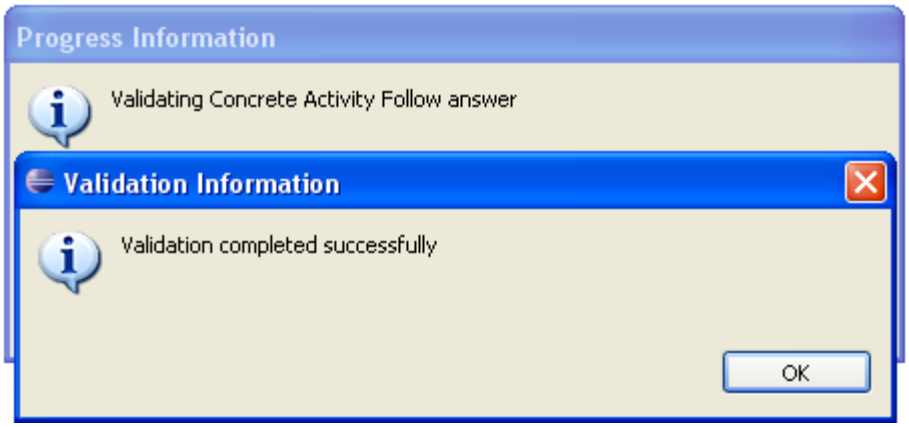

Figure 3.4. Exemple de test positif

Les actions concrètes de l'activité *Follow answer* font appel à quatre opérations critiques : *getDate()*, *getPlace()*, *getTime()* et *applyChange()* de la classe *Meeting*. Les quatre opérations sont associées à la permission *InitiatorMeeting* qui autorise le rôle *Initiator* affecté à l'activité *Follow answer* de les exécuter. La précondition locale *FA-MeetingReadUpdate* gardant ces actions correspond à la contrainte d'autorisation *IM-authConstraint* rattachée à la permission *InitiatorMeeting*.

3.5.3 Tests négatifs

Les tests négatifs permettent de voir la réaction de notre outil face à des instances incohérentes. Ils doivent se terminer par un message d'erreur qui signale les invariants violés. Pour chaque règle de cohérence, nous avons défini un ensemble de cas de test négatifs afin de tester si notre outil réagit vis-à-vis de toutes les propriétés invariantes.

Nous allons fait des changements sur le modèle d'activité concrète de la figure 2.6 et le modèle SecureUML de la figure 2.4 pour spécifier quelques exemples de cas de test négatifs. Les instances correspondant à ces modèles sont présentées dans la figure 3.2.

3.5.3.1 Invariant xMissingPermissionToLegitimateUser

Un exemple de cas de test de cet invariant est d'enlever l'action *Meeting:read* de la permission *InitiatorMeeting* dans le modèle SecureUML de la figure 2.4. La figure 3.5 montre les messages d'erreurs affichés par notre outil après le changement et la validation des modèles.

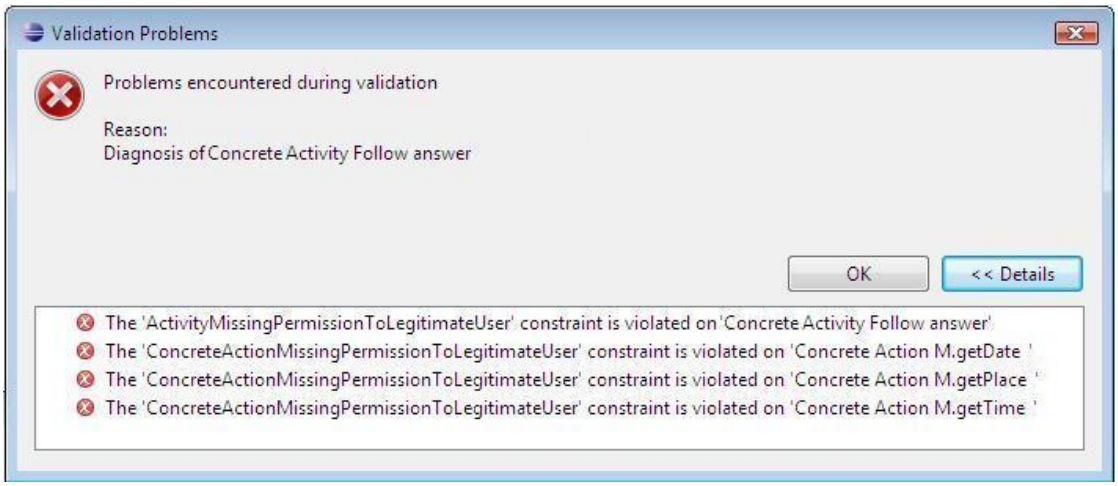

Figure 3.5. Exemple de cas de test négatif des invariants xMissingPermissionToLegitimateUser

Après la modification, l'action *Meeting:read* sera associée seulement à la permission *ParticipantMeeting*. Elle ne donne donc qu'aux utilisateurs du rôle *Participant* le droit de lire les données de la classe *Meeting.* Cependant dans le modèle d'activité *Follow answer* de la figure 2.6, un utilisateur assigné au rôle *Initiator* doit exécuter les opérations de lecture sur les données la classe *Meeting*. Dans ce cas, les opérations *getDate(), getPlace()* et *getTime()* de la classe *Meeting* sont bloquées par le modèle SecureUML et autorisées par le modèle BAAC@UML.

3.5.3.2 Invariant MissingPermissionAuthorizationConstraint

Un exemple de test négatif de cet invariant est d'enlever la contrainte d'autorisation *IM-authConstraint* rattachée à la permission *InitiatorMeeting* dans le modèle SecureUML de la figure 2.4. Après le changement et la validation, les messages de la figure 3.6 sont affichés par notre outil.

Ce changement dans le modèle SecureUML engendre une violation d'une propriété invariante. Dans le modèle d'activité *Follow answer* de la figure 2.6, l'accès aux opérations *getDate(), getPlace(), getTime() et applyChange()* de la classe *Meeting* est préconditionné par la réalisation d'une contrainte contextuelle exprimée par la

précondition de contrôle d'accès *FA-MeetingReadUpdate* ; Seul le créateur de *Meeting* peut exécuter ces opérations. Cependant dans le modèle SecureUML de la figure 2.4, aucune contrainte d'autorisation n'est exigée. Dans cette règle, nous considérons le fait qu'il y a une stricte correspondance entre les contraintes d'autorisation et les préconditions de contrôle d'accès. On peut avoir une autre vision où les préconditions de contrôle d'accès soient plus exigeantes.

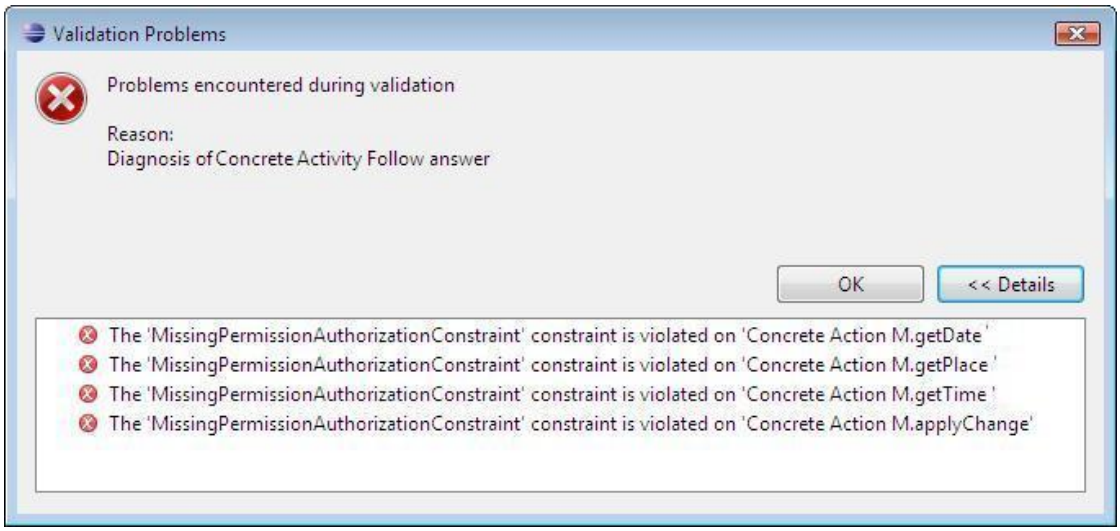

Figure 3.6. Exemple de cas de test négatif de l'invariant MissingPermissionAuthorizationConstraint

3.5.3.3 Invariant NotConformanceOfAuthorizationConstraint

Un exemple de cas de test de cette règle est d'affecter une autre contrainte d'autorisation A*C2* à la permission *InitiatorMeeting.* Les messages d'erreurs de la figure 3.7, affichés par notre outil après le changement et la validation, indiquent des violations qui portent sur la conformité des contraintes contextuelles.

Figure 3.7. Exemple de cas de test négatif de l'invariant NotConformanceOfAuthorizationConstraint

Dans ce cas, les contraintes contextuelles pré-conditionnant l'exécution des opérations *getDate()*, *getPlace()*, *getTime()* et *applyChange()* de la classe *Meeting* ne sont pas conformes dans les deux modèles de contrôle d'accès BAAC@UML et SecureUML.

3.5.3.4 Invariant MissingConcreteActionLACPC

Pour tester cet invariant, nous allons éliminer le lien entre l'action concrète *M.getPlace* et la précondition locale de contrôle d'accès *FA-MeetingReadUpdate* dans le modèle d'activité *Follow answer* de la figure 2.6. Ce changement va créer un cas d'incohérence avec le modèle SecureUML. L'accès à l'opération fonctionnelle *getPlace()* de la classe *Meeting* est conditionné par une contrainte contextuelle car cette opération est associée à des permissions rattachées à des contraintes d'autorisation dans le modèle SecureUML de la figure 2.4. Cependant dans le modèle BAAC@UML de la figure 2.6, aucune garde n'est associée à l'action concrète *M.getPlace* qui fait appel à l'opération *getPlace()* après le changement. Le message d'erreur de la figure 3.8 est affiché par notre outil après le changement et la validation des instances.

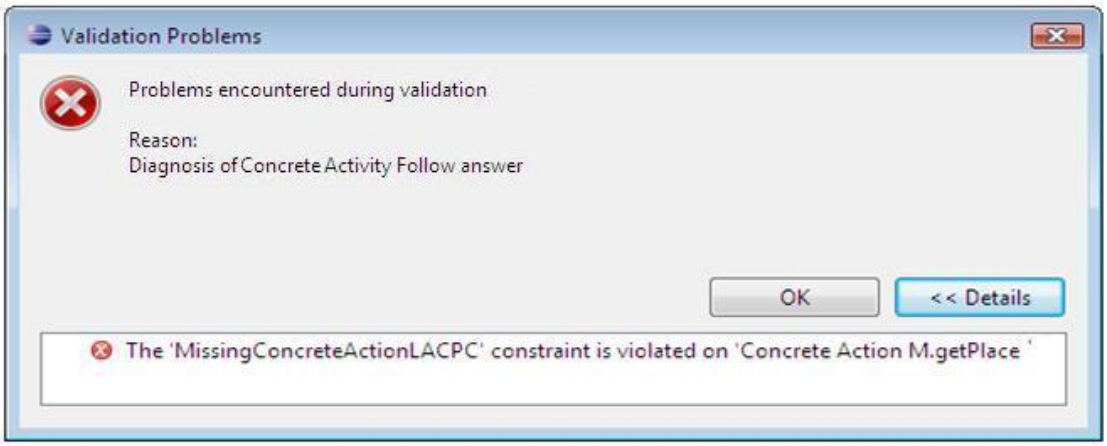

Figure 3.8. Exemple de cas de test négatif de l'invariant MissingConcreteActionLACPC

3.5.4 Bilan de la validation

La validation des règles OCL présentées dans la section 3.4 s'est basée d'une part sur la relecture attentive de ces contraintes et d'autre part sur leur expérimentation sur notre étude de cas.

Chacun des six diagrammes d'activités décrivant les six cas d'utilisation de la figure 2.2 (chapitre 2) a constitué un test positif. Ensuite nous avons introduit systématiquement des mutations de ces diagrammes pour falsifier chacun des cinq invariants. Pour chaque diagramme d'activités nous avons créé dix tests négatifs. Cette validation a été complétée par des cas de test dont les instances ne sont pas conformes au méta-modèle.

Au total soixante tests négatifs et six tests positifs ont été réalisés. Leur caractère systématique nous donne confiance dans la correction des cinq invariants et de notre outil *ACP-Consistency* mettant en œuvre ces règles.

3.6 Mise au point de l'étude de cas

Notre outil *ACP-Consistency* permet de signaler tout type d'incohérence entre les instances correspondant aux modèles SecureUML et nos modèles d'activités concrètes. Dans cette section, nous utilisons cet outil pour mettre au point l'exemple d'organisation de réunions. Plusieurs erreurs de cohérence ont été détectées et corrigées après la création et la validation des instances correspondant aux modèles BAAC@UML et aux modèles SecureUML du système d'organisation de réunions. Les modèles BAAC@UML spécifient le contrôle d'accès aux différentes activités concrètes décrivant les 6 cas d'utilisation du système. Les modèles SecureUML contrôlent l'accès aux différentes classes du système.

Notre démarche de validation de la cohérence entre les modèles de contrôle d'accès BAAC@UML et SecureUML passe par deux étapes. La première consiste à détecter les erreurs de cohérence à l'aide de notre outil *ACP-Consistency* et la deuxième fait les corrections nécessaires sur les modèles erronés.

3.6.1 Détection des incohérences

Nous présentons ici un exemple d'incohérences trouvées entre le modèle d'activité concrète *Follow answer* présenté dans la figure 2.6 et le modèle SecureUML de la figure 3.9 qui exprime le contrôle d'accès à la classe *Invitation* (voir diagramme de classes de la figure 2.3).

Dans la figure 3.9, la classe *Invitation* est protégée par deux permissions. La première *InitiatorInvit* autorise un utilisateur assigné au rôle *Initiator* à modifier les informations d'une invitation pour autant qu'il soit le créateur de la réunion concernée par cette invitation. La deuxième *ParticipantInvit* exprime que seul un utilisateur assigné au rôle *Participant* peut lire les invitations. La contrainte d'autorisation rattachée à cette permission restreint la lecture à la seule personne invitée.

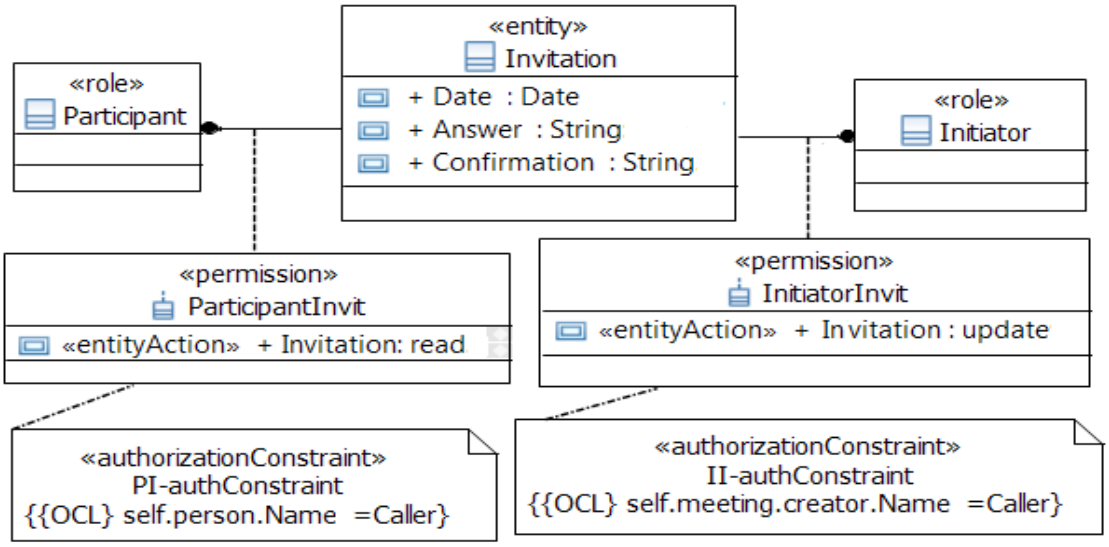

Figure 3.9. Le modèle SecureUML de contrôle d'accès à la classe Invitation

Après la création et la validation des instances correspondantes aux modèles BAAC@UML (figure 2.6) et SecureUML (figure 3.9), les messages de la figure 3.10 sont affichés par notre outil. Ils montrent trois cas de violation.

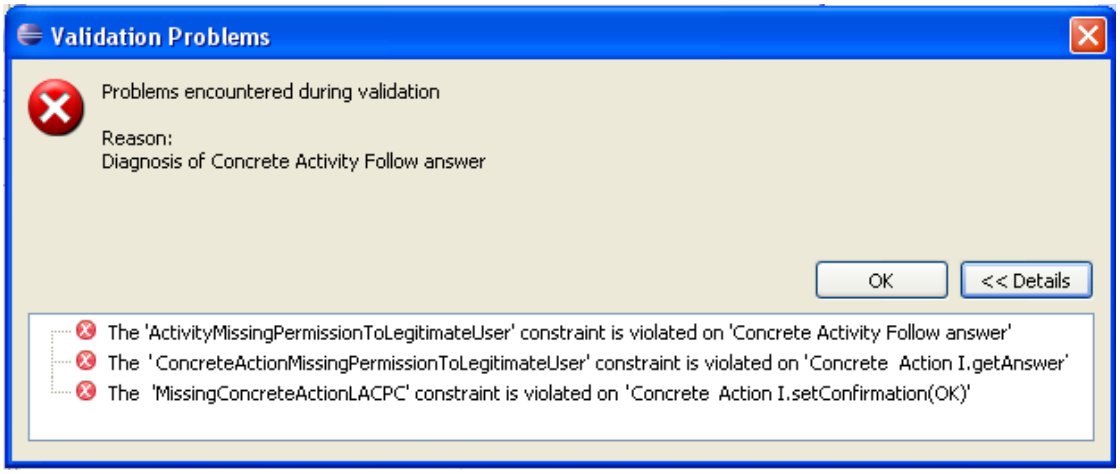

Figure 3.10. Exemple d'une détection d'incohérences

Le premier message correspond à l'invariant *ActivityMissingPermissionTo LegitimateUser*. Comme nous l'avons vu dans la section 3.4.2, il indique que l'activité concrète *Follow answer* inclut au moins une action qui fait appel à une opération non autorisée, par les permissions SecureUML de la figure 3.9, pour le rôle *Initiator*.

Le deuxième message concerne l'invariant *ConcreteActionMissingPermissionTo LegitimateUser* qui précise la contrainte précédente. Il identifie l'action non autorisée

de l'activité *Follow answer*: il s'agit de l'action *I.getAnswer*. Comme nous l'avons vu dans la section 3.4.2, la permission *ParticipantInvit* de la figure 3.9 ne donne qu'aux utilisateurs assignés au rôle *Participant* le droit de lire les données de la classe *Invitation*. Cependant dans le modèle d'activité *Follow answer* de la figure 2.6, un utilisateur assigné au rôle *Initiator* doit lire la réponse de l'invitation (L'attribut *Answer* de la classe *Invitation*). Dans ce cas l'opération *getAnswer()* est bloquée par le modèle statique SecureUML et autorisée par le modèle dynamique exprimé par notre profil.

Le dernier message correspond à l'invariant *MissingConcreteActionLACPC* qui n'est pas respecté par l'action *I.setConfirmation(OK)*. En effet, comme nous l'avons vu dans la section 3.4.3, la permission *InitiatorInvit* de la figure 3.9 est associée à une contrainte d'autorisation qui n'autorise que l'initiateur créateur de la réunion concernée par une invitation à modifier ses informations. Cependant dans le modèle de l'activité concrète *Follow answer* de la figure 2.6, aucune garde n'est associée à l'action *I.setConfirmation(OK)*.

3.6.2 Correction des erreurs de cohérence

Pour éviter ces incohérences, nous avons fait des corrections sur les modèles BAAC@UML et SecureUML. La figure 3.11 présente le modèle SecureUML de contrôle d'accès à la classe *Invitation* (figure 3.9) après correction.

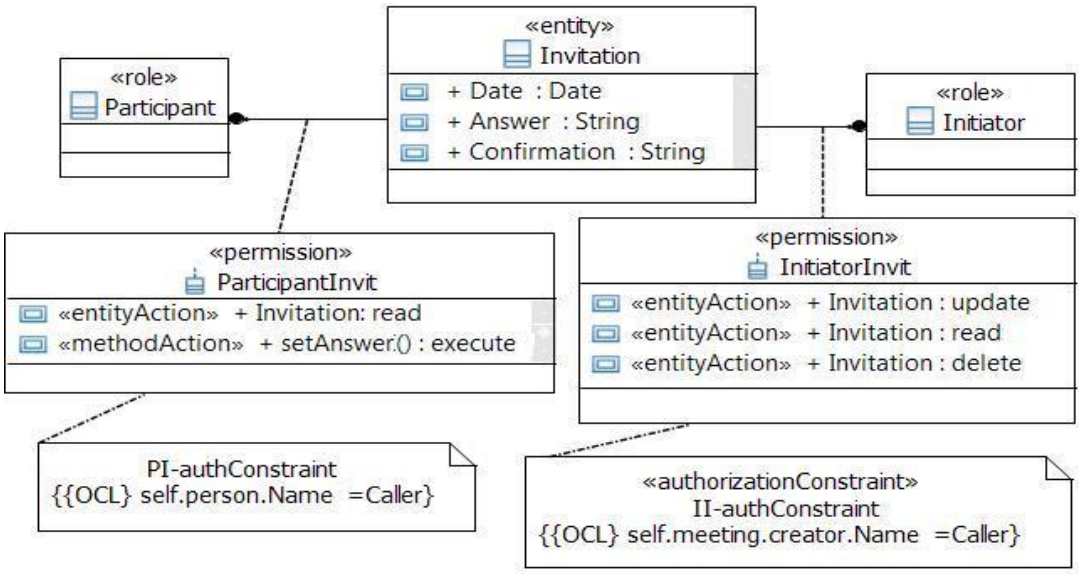

Figure 3.11. Correction du modèle SecureUML de contrôle d'accès à la classe Invitation

Deux actions ont été ajoutées à la permission *InitiatorInvit* : *«entityAction» Invitation:read* et *«entityAction» Invitation:delete* qui autorisent respectivement le rôle *Initiator* à lire et à supprimer les invitations qu'il a créées. L'action *«MethodAction» setAnswer():execute* ajoutée à la permission *ParticipantInvit* permet au rôle *Participant* de modifier la réponse de ses invitations.

Pour l'activité *Follow answer* de la figure 2.6, les trois actions : *I.getAnwer*, *I.setConfirmation(OK)* et *I.delete* sont rattachées à la précondition locale de contrôle d'accès définie par l'expression OCL *I.meeting.Creator.Name=Caller*. La figure 3.12 présente l'activité *Follow answer* après correction.

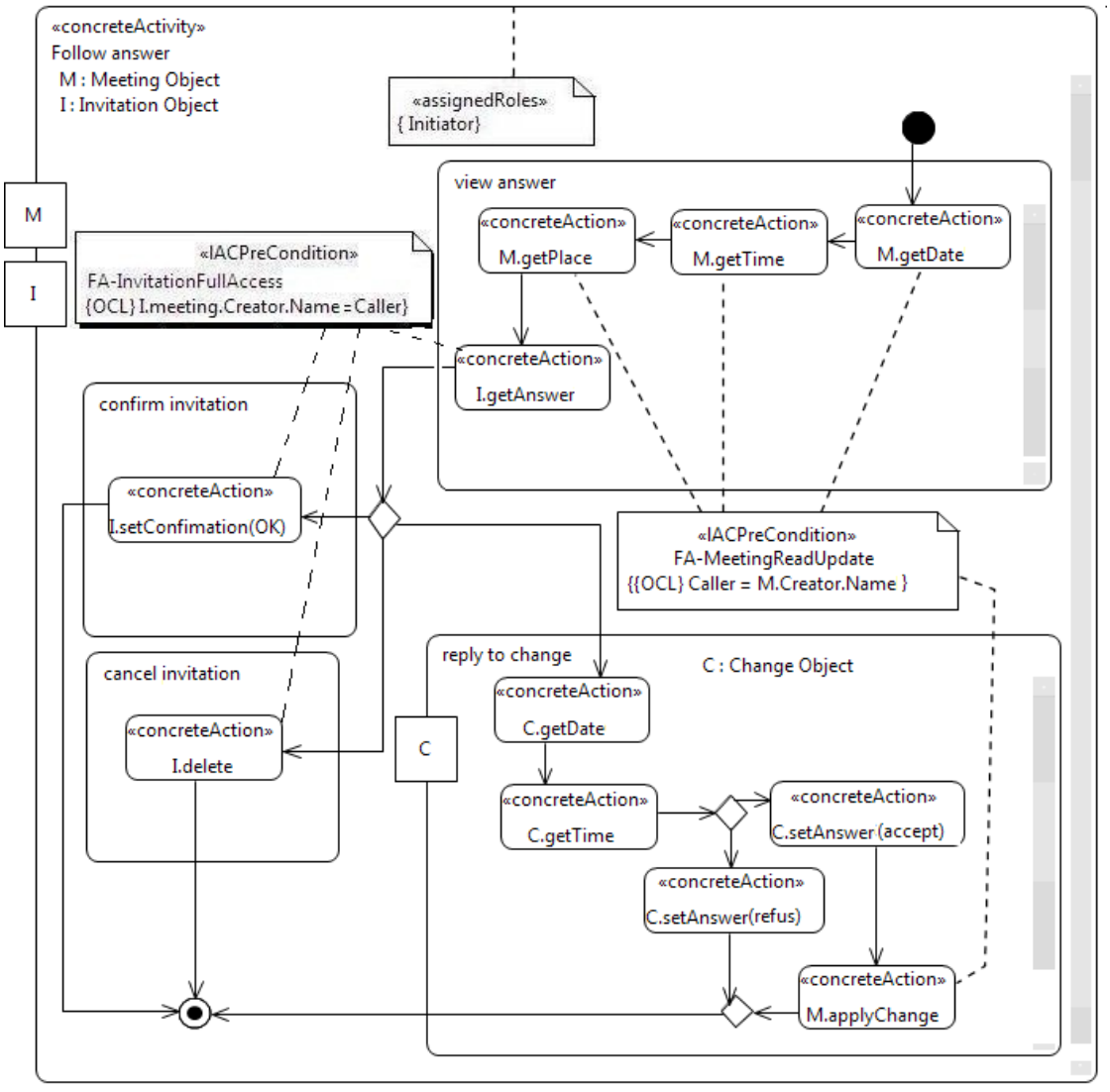

Figure 3.12. Correction du modèle BAAC@UML de contrôle d'accès à l'activité concrète Follow answer

D'autres corrections, sur les autres diagrammes d'activités concrètes et sur les modèles SecureUML, ont été réalisées.

3.7 Conclusion

L'expérience montre que de très nombreuses fautes de modélisation sont décelables à travers la détection d'incohérences (Lange et al., 2003). Dans ce chapitre, nous avons complété notre profil BAAC@UML par la spécification d'un ensemble de règles de cohérence, exprimées en OCL, qui vérifie la conformité de ses modèles par rapport aux modèles SecureUML. Nous avons traité la cohérence inter-modèles en considérant la conformité des rôles et des contraintes contextuelles.

Notre travail a été supporté par l'outil *ACP-Consistency* qui permet de signaler toute contradiction entre les représentations statiques et dynamiques de politiques RBAC. Cet outil nous a permis de valider les différentes règles de cohérences. Une centaine de tests entre positifs et négatifs ont été exécutés sur les différents modèles SecureUML et BAAC@UML, en vue de vérifier les 5 invariants OCL.

ACP-Consistency nous a également permis de mettre au point le système d'organisation de réunions et de corriger les erreurs de cohérence détectées dans les modèles de contrôle d'accès des vues statique et dynamique.

Peu de travaux ont abordé le problème de la cohérence entre les profils de contrôle d'accès. (Matulevicius, Dumas, 2011) présente des règles de transformation entre SecureUML et UMLsec. (Montrieux et al., 2011) propose deux nouveaux profils pour la spécification de politiques RBAC au niveau des diagrammes de classes et d'activités, et également un outil qui vérifie la cohérence entre eux. Dans ce travail, nous avons proposé des règles qui assurent la cohérence entre le profil BAAC@UML que nous proposons pour spécifier le contrôle d'accès aux activités concrètes et SecureUML qui est un profil bien adapté pour spécifier une politique RBAC dans une vue statique.

Dans le chapitre suivant, nous proposons la traduction des modèles BAAC@UML et SecureUML en B pour la vérification des autres cas d'incohérences et pour la validation de la politique de sécurité exprimée par ces modèles.

Chapitre 4

Spécification et validation formelles de modèles BAAC@UML

4.1 Introduction

L'augmentation des risques inhérents aux systèmes d'information fait que la spécification de politiques de sécurité doit suivre une démarche précise. Le profil BAAC@UML proposé dans le cadre de notre étude est basé sur le langage UML dont la sémantique n'est pas entièrement formelle. En effet, ce langage présente une notation qui se veut structurante, intuitive et facile à appréhender. En revanche, la sémantique d'UML est souvent qualifiée de floue du fait qu'elle manque de bases mathématiques (Idani, 2006). Pour répondre à ce besoin, l'utilisation des méthodes formelles, comme par exemple la méthode B (Abrial, 1996), est considérée comme l'un des principales solutions.

Dans ce chapitre, nous proposons de compléter les modèles graphiques de contrôle d'accès, exprimés au moyen de notre profil BAAC@UML, par une spécification B. D'une part, les modèles graphiques permettent d'abstraire une politique RBAC, ce qui facilite sa compréhension, et d'autre part, leur spécification en B fournit une base pour la vérification et la validation (V&V) de la politique RBAC. Pour ce faire, nous utilisons la plateforme B4MSecure[8]. Cet outil, fondé sur l'approche IDM, permet la transformation automatique en B de diagrammes de classes fonctionnels et de modèles SecureUML dans le but de les valider formellement. Nous proposons une extension de cette plateforme pour la dérivation de nos modèles BAAC@UML en B.

Notre approche de traduction de modèles d'activités BAAC@UML en spécifications B se fait en trois étapes : la première consiste à traduire le diagramme de classes fonctionnel, la deuxième accomplit la transformation de modèles SecureUML et enfin la dernière étape produit un modèle B à partir des modèles BAAC@UML. Ce modèle B formalise les activités et leur affectation aux rôles ainsi que les différentes préconditions de contrôle d'accès associées à leurs actions.

La spécification B des modèles BAAC@UML définit un cadre rigoureux favorisant la V&V de la politique de sécurité à l'aide des outils d'animation et de preuve dédiés de la méthode B. Dans ce travail, nous utilisons la preuve pour prouver la conformité de certaines propriétés de nos modèles BAAC@UML. Ainsi, nous utilisons la technique d'animation pour tester les règles de la politique RBAC exprimée par les modèles BAAC@UML et pour vérifier la cohérence sémantique entre les modèles BAAC@UML et les modèles SecureUML.

Les travaux de ce chapitre sont présentés dans *IEEE Tenth International Conference on Research Challenges in Information Science (RCIS2016)* (http://ieeexplore.ieee.org/document/7549284/?arnumber=7549284).

[8] http://b4msecure.forge.imag.fr/

Ce chapitre est organisé de la manière suivante : la section 4.2 introduit les fondements de la méthode B, les approches d'intégration d'UML et B et les différentes techniques de test de spécifications B. La traduction des modèles BAAC@UML en B sera décrite dans la section 4.3. Dans la section 4.4, nous tirerons profit des outils dédiés de la méthode B pour la validation de modèles B issus des diagrammes BAAC@UML. La dernière section dresse les conclusions de ce chapitre.

4.2 La méthode B

Les méthodes formelles, initiées par les travaux du *Programming Research Group* de l'université d'Oxford, consistent à exploiter les mathématiques pour le développement rigoureux des systèmes informatiques. Ces méthodes sont fondées sur un langage à syntaxe et à sémantique précises permettant des vérifications automatisées. Elles peuvent être utilisées de plusieurs manières et dans les différentes phases de processus de développement. (Meyer, 2001) propose une classification des méthodes formelles en quatre catégories: les méthodes algébriques basées sur les types abstraits de données (comme par exemple OBJ (Futatsugi et al., 1985)), les méthodes dynamiques qui permettent la spécification du comportement des systèmes (comme CSP(Hoare, 1978)), les méthodes basées sur la théorie des types et la logique d'ordre supérieur (comme par exemple PVS (Owre et al.,1992)) et les méthodes basées sur la théorie ensembliste (comme par exemple B (Abrial, 1996) et Z (Spivey, 1989)).

Dans ce travail, nous nous sommes basés sur la méthode B, fondée sur la théorie des ensembles, pour la spécification formelle des modèles UML de contrôle d'accès et la validation formelle de politiques de sécurité exprimées au moyen de ces modèles. La méthode B a été proposée par *Jean-Raymond Abrial* vers le milieu des années 80 pour le développement structuré, précis, modulaire et successif de systèmes. La notion de *machine abstraite* représente l'élément fondamental de structuration des spécifications, la précision est assurée au moyen des prédicats qui associent des types aux données de système et enfin les clauses de composition et de partage facilitent la décomposition des systèmes complexes en modules.

Cette section met l'accent sur les éléments et les techniques de la méthode B utilisées dans notre approche de spécification et de validation formelles de modèles UML exprimant une politique RBAC. Nous présentons la notion de machine abstraite et de ses composants, les approches d'intégration entre UML et B et les techniques de test de spécifications B.

4.2.1 La machine abstraite et ses composants

La machine abstraite est un modèle mathématique qui exprime les propriétés du système. Ce modèle est constitué des états que le système peut prendre, des relations

d'invariance qui sont maintenues lorsque le système passe d'un état à un autre, et des opérations permettant de fournir un service aux utilisateurs du système (Behnia, 2000). Ainsi, la machine abstraite encapsule la spécification des aspects statiques et dynamiques du système. La spécification statique est représentée par des variables qui définissent l'état du système modélisé. La spécification dynamique définit des opérations qui permettent de manipuler ces variables et faire évoluer l'état du système.

Partie entête
MACHINE
- Nom de la machine
- Paramètres de la machine

CONSTRAINTS
- Contraintes sur les paramètres

Partie composition

Partie statique

Partie dynamique

END

Figure 4.1. Structure d'une machine abstraite

De façon générale, la machine abstraite est composée de quatre parties (voir la figure 4.1) : l'entête, les compositions, les déclarations (la partie statique) et une partie exécutive (la partie dynamique). La partie entête marque le début de la spécification de la machine abstraite et permet de l'identifier. Elle contient la clause MACHINE décrivant le nom de la machine suivi éventuellement de paramètres, ainsi que la clause CONSTRAINTS où il s'agit de caractériser ces paramètres qui peuvent être des valeurs scalaires, des types prédéfinis, ou bien des ensembles. La partie des clauses de composition permet d'établir des liens avec d'autres machines ou composants B. La partie déclaration définit l'état et le contexte de la machine abstraite. Enfin, la partie exécutive initialise la machine et spécifie ces opérations. Le "END" final marque la fin de la machine. Les compositions, les déclarations et les opérations seront introduites plus en détail dans les sections 4.2.1.1, 4.2.1.2 et 4.2.1.3 respectivement.

4.2.1.1 La partie composition

La figure 4.2 présente les différentes clauses de composition de la méthode B. L'introduction de la notion de composition en B est une réponse aux besoins de modularité et de développement progressivement perçus comme étant des solutions

efficaces pour maîtriser la complexité d'un système et faciliter sa compréhension. La littérature distingue deux types de compositions en B : *verticale* et *horizontale*.

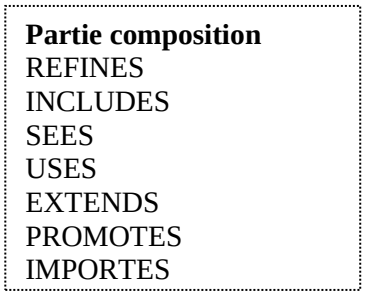

Figure 4.2. Les clauses de composition d'une machine abstraite

La composition *verticale* est réalisée par le mécanisme de raffinement qui considère différents niveaux d'abstraction du système. L'objectif est le développement incrémental et successif qui passe d'une spécification de haut niveau à une spécification concrète. Le raffinement d'une machine par une autre est réalisé au moyen de la clause REFINES qui permet de préciser deux aspects de la machine abstraite raffinée : son état, et son comportement dynamique. Lors du raffinement, l'état peut être détaillé en précisant avec une granularité plus fine les variables de la machine, ou en ajoutant de nouvelles variables. De manière duale, le comportement dynamique de la machine est précisé en modifiant les opérations conformément aux anciennes et nouvelles variables (Belhadj, 2012).

La composition *horizontale* permet la décomposition du système en modules dans le but de construire une spécification en le réduisant à des spécifications plus petites. La méthode B propose les clauses INCLUDES, SEES, USES, EXTENDS, PROMOTES et IMPORTS pour définir les règles de visibilité entre les machines abstraites. (Potet, 2002) distingue deux modes de visibilité selon l'accès aux variables de l'extérieur : *ouvert* et *semi-ouvert*. Dans le cadre du mode ouvert, les variables de la machine appelée par les clauses de composition sont visibles sans contrainte et accessibles à travers les opérations de la machine appelante. En revanche, les variables de la machine appelée par les clauses de composition ne sont visibles de l'extérieur qu'en lecture dans le mode semi-ouvert.

La clause INCLUDES est un exemple de mode ouvert ; elle permet de rendre visible, dans la machine incluante, les données et les opérations de la machine incluse (Idani, 2006). Les clauses SEES et USES sont des exemples de mode semi-ouvert. Une machine abstraite qui en voit une autre, par la relation SEES, peut en consulter les données et en utiliser les opérations qui ne modifient pas les données. Lorsqu'une machine abstraite utilise, avec la relation USES, une autre machine abstraite, elle peut

en utiliser les données, y compris les paramètres formels, dans ses invariants et ses opérations sans les modifier (Bolusset, 2004).

4.2.1.2 La partie statique

Cette partie est représentée par la figure 4.3. Elle déclare les données de la machine abstraite qui regroupe des ensembles, des constantes et des variables. Les ensembles sont définis dans la clause SETS, les constantes dans la clause CONSTANTS et les variables dans la clause VARIABLES.

Partie statique
SETS
- Déclaration d'ensembles

CONSTANTS
- Déclaration de constantes

PROPERTIES
- Propriétés des constantes

VARIABLES
- Variables (état)

INVARIANT
- Invariant (caractérisation de l'état)

DEFINITIONS
- Définitions (*prédicats du premier ordre)*

Figure 4.3. La partie statique d'une machine abstraite

Il est également possible de compléter ces données en utilisant les clauses PROPERTIES, INVARIANTS et DEFINITIONS. Dans la clause PROPERTIES, on peut définir un ensemble de prédicats décrivant les propriétés des constantes. On peut aussi spécifier des invariants explicitant des propriétés qui doivent toujours être satisfaites par l'état de la machine dans la clause INVARIANTS. La clause DEFINITIONS contient une liste d'abréviations pour un prédicat, une expression ou une substitution. Les définitions peuvent être utilisées dans la suite de la machine et sont locales à la machine où elles sont définies (Nguyen, 1998).

Les données déclarées dans cette partie sont spécifiées en utilisant des formules de la logique du premier ordre et des concepts mathématiques de la théorie des ensembles comme les ensembles, les relations, les fonctions, les séquences, etc. Ainsi, les données peuvent être associées à un type en utilisant, par exemple, des prédicats de typage comme l'appartenance (∈), l'inclusion (⊆) et l'égalité (=). La clause INVARIANTS permet de typer les variables.

(ClearSy, 2014b) considère le typage comme un mécanisme de vérification statique des données. Il distingue deux catégories de types en B. Les types de base, comme les

entiers (Z), les booléens (BOOL), les chaines de caractères (STRING) et les ensembles abstraits et énumérés définis au niveau de la clause SETS. La deuxième catégorie de types est définie à l'aide d'un constructeur de type, comme l'ensemble des parties finies d'un ensemble et le produit cartésien entre deux ensembles.

4.2.1.3 La partie dynamique

La partie exécutive de la machine abstraite est définie dans la figure 4.4. Elle décrit l'évolution de l'état de la machine et est représentée par les opérations (Clause OPERATIONS) et les initialisations (Clause INITIALISATION).

Partie dynamique
INITIALISATION
- Initialisation de l'état → *substitutions généralisées*

OPERATIONS
- Opérations → *substitutions généralisées*

Figure 4.4. La partie dynamique d'une machine abstraite

La clause « INITIALISATION » définit les valeurs initiales des différentes variables de la machine. Toute variable doit être initialisée et cette initialisation doit satisfaire l'invariant de la machine.

La clause « OPERATIONS » contient la liste des opérations de la machine qui permettent d'accéder à l'état du système et de le modifier dans la limite des invariants. Une opération peut contenir une précondition qui exprime les conditions indispensables pour pouvoir invoquer l'opération. Elle se compose de deux parties : un *entête* et un *corps*. L'entête de l'opération est constitué d'un identificateur désignant le nom de l'opération et des éventuels paramètres d'entrée et de sortie. Le corps de l'opération est constitué d'un ensemble d'instructions qui manipulent les variables de la machine.

La spécification des opérations et de l'initialisation est décrite par le langage des substitutions généralisées. Ce langage est une notation spécifique à la méthode B qui représente une transition sur l'état du modèle B. (Truong, 2006) distingue deux types de substitutions généralisées : *élémentaire* et *complexe*. Il existe deux substitutions élémentaires : l'une, la substitution identité, notée *skip*, ne modifie pas l'état courant, et l'autre la substitution simple, notée par $x := E$, le modifie. $x := E$ permet d'assigner la valeur de l'expression E à la variable x. Les substitutions complexes sont construites à partir de ces deux substitutions élémentaires et des opérateurs de composition tels que la séquence (;), le parallélisme (||), le choix (if...then...end), etc.

4.2.2 Approche d'intégration entre UML et B

L'approche de couplage entre UML et les méthodes formelles, appelée aussi *approche mixte*, est un sujet étudié depuis les années 1990. Le but de cette approche est de donner une sémantique basée sur des fondements mathématiques aux modèles UML. Différentes méthodes ont été utilisées pour la formalisation des modèles UML. Par exemple : (Kim, 2001) présente un travail autour de la méthode Object-Z. (Borges, Mota, 2007) propose l'intégration du diagramme de classes d'UML et le langage formel *OhCircus*. Dans notre étude, nous allons utiliser la méthode B pour la formalisation de nos diagrammes d'activités de contrôle d'accès. L'intégration d'UML et de B est motivée par le souhait de pouvoir les utiliser conjointement dans un développement intégrant à la fois précision et structuration. D'une part, UML possède un aspect structurel, intuitif et synthétique grâce à l'utilisation de ses diagrammes. D'autre part, la méthode B permet de construire des spécifications précises et d'identifier les ambiguïtés mieux qu'en UML (Ayed et al., 2014).

(Meyer, 2001) et (Truong, 2006) définissent quatre approches d'intégration entre les notations semi-formelles et les méthodes formelles. L'intégration *par adjonction* consiste à remplacer une partie de la description informelle du modèle par une expression formelle. L'intégration *par définition* vise à faire de la notation semi-formelle une véritable notation formelle en définissant une sémantique pour ses concepts et des outils permettant la manipulation directe de ses modèles. L'intégration *par extension* a comme objectif d'enrichir les langages de spécification formels par les concepts des langages semi-formels. L'intégration *par dérivation* permet de traduire les modèles semi-formels en des modèles formels ou inversement. Dans notre travail, nous allons nous concentrer sur la technique de dérivation. Plusieurs travaux s'inscrivent dans cette approche. Parmi ces travaux : celui de (Meyer, 2001) porte sur la dérivation des concepts structurels de diagrammes de classes et de diagrammes d'état-transitions. (Ledang, 2002) présente une approche pour la formalisation des concepts comportementaux comme les évènements et les opérations. (Marcano, 2002) transforme les expressions OCL en B. (Laleau, 2000) propose la dérivation du méta-modèle UML en B pour exprimer la sémantique de diagrammes de classes.

Selon (Facon, Laleau, 1995) et (Idani, 2006), il existe deux approches de dérivation de spécification semi-formelle telle que UML, en spécification formelle: *l'approche interprétée* et *l'approche compilée*.

(A1) *L'approche interprétée*, appelée aussi *par méta-modélisation*, se fait en deux couches. La première couche commence par la formalisation en B des concepts du méta-modèle d'UML, ensuite, la deuxième couche traduit et injecte la spécification B des éléments propres du modèle UML dans la spécification B de la première couche. (Laleau, 2002), par exemple, est basé sur cette approche.

(A2) *L'approche compilée*, appelée aussi *par traduction*, définit un ensemble de règles permettant de générer directement des spécifications B à partir de modèles UML. La plupart des travaux de dérivation de UML vers B sont inscrits dans cette approche, comme par exemple, (Meyer, 2001), (Mammar, 2002), (Snook et al., 2006) et (Idani, 2006). Dans notre travail, nous allons adopter l'approche compilée pour la traduction de nos modèles d'activités de contrôle d'accès vers B.

Il existe plusieurs outils qui supportent la traduction des modèles UML en B. Nous distinguons, par exemple, UML2B (Hazem et al., 2004), UML2SQL (Laleau et al., 2000), (Mammar et al., 2004), U2B (Snook et al., 2004) et ArgoUML+B (Ledang et al., 2003). Dans le cadre de notre étude, nous utilisons la plateforme B4MSecure (Idani et al., 2015) dédiée à la dérivation en B des diagrammes UML fonctionnels ainsi que des diagrammes UML exprimant des politiques de contrôle d'accès.

4.2.3 Test de spécifications B

Dans le cadre d'un développement formel en B, l'utilisation des techniques de vérification et de validation (V&V) est centrale. Ces techniques garantissent la validité des spécifications et permettent d'éviter la majeure partie de la phase de tests du logiciel obtenu (Behnia, 2000). Dans le cadre de notre étude, le test de spécifications B formalisant les modèles BAAC@UML vérifie que ces modèles sont cohérents avec les modèles SecureUML et qu'ils représentent les règles de la politique de contrôle d'accès du système réel. Il vise à révéler les fautes dues à une mauvaise modélisation de la politique RBAC et à corriger certaines propriétés du système. Dans cette section, nous présentons les deux techniques de V&V de spécifications B que nous allons utiliser dans notre approche : *la preuve* et *l'animation*.

4.2.3.1 Technique de preuve

La preuve assure la correction de certaines propriétés de spécifications B. Les conditions à satisfaire pour garantir ces propriétés sont appelées *obligations de preuve*. (ClearSy, 2014b) définit une obligation de preuve comme étant « *une formule mathématique à démontrer afin d'assurer qu'un composant B est correct. La théorie B indique quelles sont les obligations de preuve à démontrer pour assurer la correction d'un composant B donné. Dans cette optique, les obligations de preuve sont une aide au processus de vérification* ».

Les obligations de preuve accompagnent la spécification B. Elles démontrent certaines propriétés de la spécification et vérifient de manière rigoureuse les phases de développement formel. Elles sont générées systématiquement et automatiquement par les outils de la méthode appelés aussi *générateurs d'obligations de preuves*. Il s'agit

de *la preuve de correction de la machine abstraite* qui sert à prouver les propriétés d'invariance d'une machine que le système doit préserver quelle que soit son évolution. Il s'agit également de *la preuve de correction du raffinement* vis-à-vis de ses versions plus abstraites. Le raffinement doit préserver les propriétés déjà prouvées dans les spécifications plus abstraites.

Soient *P* et *H* des prédicats, une obligation de preuve a la forme de la formule F1. Elle signifie qu'il faut démontrer le but *P* sous l'hypothèse *H*. L'hypothèse représente des informations sur le contexte du composant B. Ces informations concernent les propriétés *PR* (clause PROPERTIES) et les contraintes *C* (clause CONSTRAINTS) de la spécification. Les clauses de composition peuvent ajouter des informations dans le contexte.

$$H \Rightarrow P \quad \text{(F1)}$$

Les obligations de preuve de correction de la machine abstraite consistent en deux types :

- Le premier type est défini par la formule F2 ; il démontre que l'invariant *I* de la machine abstraite est respecté par la substitution d'initialisation *Si*. Il s'agit de prouver que l'exécution de la substitution d'initialisation *Si* à partir d'un état satisfaisant le contexte *(PR ∧ C)* aboutit à un état qui satisfait l'invariant *I* de la machine.

$$PR \wedge C \Rightarrow [Si]\, I \quad \text{(F2)}$$

- Le deuxième type est défini par la formule F3. Il prouve que chaque opération de la machine abstraite conserve l'invariant. Etant donné que les opérations ont la forme *PRE Pr THEN So END*, l'obligation de preuve de la formule F3 est générée pour chaque opération. Sachant que l'invariant de la machine *I* et la précondition de l'opération *Pr* sont vrais pour un état satisfaisant le contexte *(PR ∧ C)* avant l'appel de l'opération, la formule F3 démontre que l'invariant reste vrai après l'exécution de la substitution de l'opération *So*.

$$PR \wedge C \wedge I \wedge Pr \Rightarrow [So]\, I \quad \text{(F3)}$$

La correction du raffinement nécessite également deux types d'obligations de preuve. Il s'agit de montrer que l'initialisation du raffinement établit l'invariant sans contredire l'initialisation de l'abstraction et que les opérations de ce composant vérifient les préconditions et l'invariant sans contredire l'opération de l'abstraction.

Il existe plusieurs outils permettant la génération des obligations de preuve et leur démonstration, comme par exemple, *B-Toolkit* (B-Core, 1996) commercialisé par *B-Core UK Ltd* et *Atelier B* (ClearSy, 2014a) de *ClearSy* qui nous allons utiliser dans notre travail.

4.2.3.2 Technique d'animation

L'animation fournit un moyen utile pour la vérification dynamique de spécifications B. L'objectif de l'animation est d'observer le comportement du système et d'analyser ses propriétés dans le but de révéler les erreurs de conception avant l'implémentation. Cette analyse permet, d'une part, de détecter certaines incohérences dans les spécifications B formalisant un modèle, et d'autre part, de vérifier la conformité de la spécification B par rapport aux besoins réels du système.

L'animation consiste à simuler l'exécution des opérations d'une spécification B, ce qui permet de changer l'état du système et de visualiser le comportement exact du système réel. Les animateurs calculent d'abord l'état initial d'une machine abstraite et ensuite affichent l'ensemble des opérations qui pouvant être exécutées. Ce sont les opérations qui vérifient les préconditions et l'invariant de la machine. (Behnia, 2000) distingue deux types d'approches d'animation de spécifications B :

- La première approche correspond à une exécution symbolique de la spécification en utilisant les techniques de programmation logique avec contraintes ensemblistes. Les variables de la spécification sont représentées par un système de contraintes et les opérations sont traduites sous forme de clauses du langage de programmation logique. Dans ce contexte, l'exécution des opérations est simulée par la réduction des contraintes. Cette technique d'animation permet de conserver l'indéterminisme et de diminuer le nombre d'états générés.
- La deuxième approche fournit des commandes permettant d'exécuter des opérations d'un composant B. Dans le cadre de cette approche, les variables d'état de la spécification sont représentées par un ensemble de valeurs et non pas par un système de contraintes. Des règles de réécriture sont utilisées pour simplifier les prédicats et les expressions en fonction de valeurs de variables. En cours d'exécution, chaque fois qu'une substitution indéterministe est rencontrée, l'animation est interrompue. L'utilisateur est alors chargé de lever l'indétermination.

Plusieurs outils d'animation de spécifications B ont été proposés dans la littérature comme par exemple *ProB* (Leuschel, Butler, 2008), *Brama* (Servat, 2006), *AnimB* (Métayer, 2010). Dans notre travail, nous allons utiliser l'outil *ProB* qui se base sur la deuxième approche d'animation présentée au-dessus.

Comme dans le cas de test de logiciels, l'animation ne peut montrer que la présence d'erreurs, mais pas leur absence (Miller, Strooper, 2001). En revanche, la spécification systématique de tests fait de l'animation un puissant moyen pour trouver des erreurs.

4.3 Traduction de modèles BAAC@UML en B

Le but de la dérivation des modèles BAAC@UML en spécifications B est de vérifier rigoureusement la politique RBAC exprimée au niveau des activités d'un BP en tirant profit des outils de la méthode B. Notre approche est basée sur la notion fondamentale de *machine abstraite,* présentée dans la section 4.2.1, qui permet de structurer les actions des différentes activités. La machine abstraite permet également de spécifier les invariants que les actions doivent respecter. Les modèles d'activités BAAC@UML seront formalisés par des machines abstraites (voir figures 4.11 et 4.13). Dans le cadre de notre approche, nous utilisons la plateforme logicielle *B4MSecure* (Idani, Ledru, 2015) dédiée à la modélisation conjointe en UML et B des aspects fonctionnels du système d'information ainsi que des politiques de contrôle d'accès.

Cette section est structurée comme suit. Nous présentons notre approche de dérivation dans la section 4.3.1 puis nous traitons de la traduction des modèles BAAC@UML en B. Notre approche commence par la traduction de modèles SecureUML (Section 4.3.2). Les modèles B générés à partir de modèles SecureUML sont ensuite utilisés dans les spécifications B des modèles BAAC@UML (Section 4.3.3).

4.3.1 Approche de traduction

L'outil *B4MSecure* a été développé dans le cadre du projet *Selkis*[9]. Cet outil permet de modéliser des diagrammes UML à l'aide de l'éditeur graphique de l'outil *Topcased* (Farail et al., 2006) et de produire des spécifications formelles B à partir de ces diagrammes. Plusieurs travaux de recherche, comme par exemple (Ledru et al., 2015) et (Milhau et al., 2011) montrent l'utilité et l'efficacité de cet outil pour la spécification en B des modèles UML dans le but de leur validation formelle.

B4MSecure assure la partie A de la figure 4.5. Il permet la traduction automatique en spécifications B des diagrammes de classes fonctionnels et des modèles SecureUML. La section 4.3.2 explique cette partie qui produit trois machines abstraites : la première (Functional) issue du diagramme de classes du système, la

[9] *ANR-08-SEGI-018*

deuxième (RBAC) et la troisième (UserAssignments) issues des modèles SecureUML. La figure 4.5 présente les dépendances entre la machine *RBAC* et les deux machines *UserAssignment* et *Functional*. La primitive INCLUDES rend visible (sans contrainte) les données et les opérations des machines *UserAssignment* et *Functional* dans la machine *RBAC*.

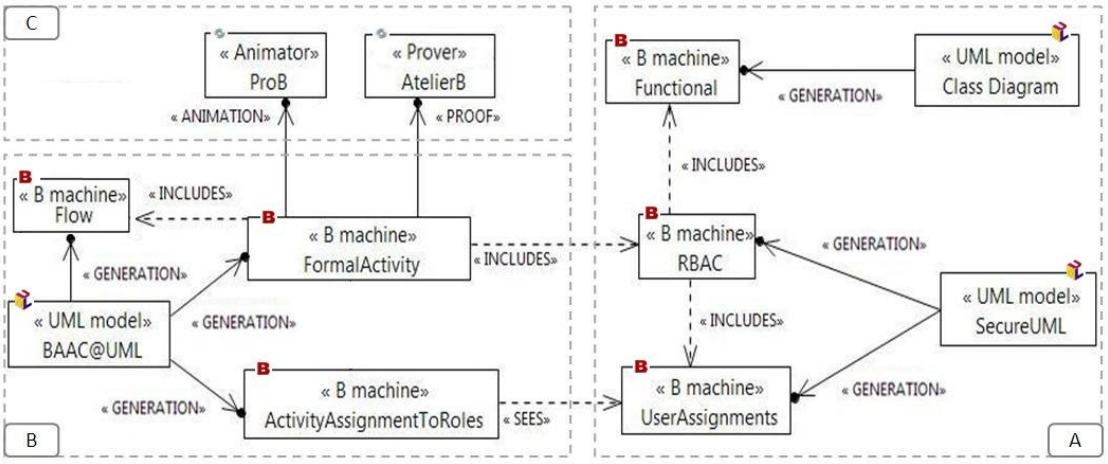

Figure 4.5. Approche de traduction en B des modèles BAAC@UML et SecureUML

Dans ce travail, nous proposons une approche pour réaliser la partie B de la figure 4.5 à l'aide de l'outil B4MSecure. La section 4.3.3 décrit cette approche qui vise la dérivation des modèles BAAC@UML en machines formelles B. Dans la partie B de la figure 4.5, notre approche résulte en trois machines abstraites *FormalActivity*, *ActivityAssignmentToRoles* et *Flow*.

- La machine *FormalActivity* représente un ensemble de machines qui spécifient les différentes activités BAAC@UML avec les transitions entre leurs actions et leurs éléments de contrôle d'accès qui vérifient les contraintes contextuelles d'autorisation et l'accès des utilisateurs aux activités. Chaque activité résulte en une seule machine.
- La machine *ActivityAssignmentToRoles* formalise l'affectation des rôles aux activités et les mécanismes d'administration de ces affectations.
- La machine *Flow* définit les éléments de spécification de flot de contrôle d'activités.

La figure 4.5 présente les dépendances entre les machines *FormalActivity*, *ActivityAssignmentToRoles* et *Flow*. Ces machines considèrent également les spécifications formelles issues de modèles SecureUML (les machines *RBAC et UserAssignment)*. La primitive INCLUDES rend visible (sans contrainte) les données et les opérations des machines *RBAC* et *Flow* dans la machine *FormalActivity*. La

primitive SEES rend visible, en lecture seulement, les données et les opérations de la machine *UserAssignment* dans la machine *ActivityAssignmentToRoles.*

La partie C de la figure 4.5 est décrite dans la section 4.4. Elle représente les activités de V&V de la politique de sécurité, exprimée au moyen de nos modèles BAAC@UML, à travers leur spécification formelle. Nous nous sommes basés sur la technique d'animation, en utilisant l'outil *ProB* (Leuschel, Butler, 2008), pour réaliser un ensemble de tests sur les machines d'activités. Nous avons également utilisé l'outil *Atelier B*[10] de la compagnie *ClearSy* pour prouver la correction de certaines propriétés des machines d'activités.

Afin d'illustrer notre approche, nous allons utiliser les exemples de notre étude de cas du système d'organisation de réunions présentés dans les chapitres précédents (le diagramme de classes fonctionnel de la figure 2.3 (Chapitre 2), les modèles SecureUML des figures 2.4 (Chapitre 2) et 3.11 (Chapitre 3), et le modèle BAAC@UML de la figure 3.12 (Chapitre 3).

4.3.2 Spécification du filtre formel RBAC

Cette section présente les grandes lignes de la traduction de la vue statique d'une politique RBAC exprimée par les modèles SecureUML en B, à l'aide de l'outil B4MSecure. Cette spécification crée un filtre qui donne le droit aux seuls utilisateurs autorisés par la politique de sécurité, d'accéder aux données du système. Les sections 4.3.2.1 et 4.3.2.2 expliquent les deux étapes nécessaires pour la construction du filtre formel RBAC. La première permet d'obtenir une unique machine abstraite à partir du diagramme de classes fonctionnel, et la deuxième utilise cette dernière machine pour construire le filtre RBAC à partir des modèles SecureUML.

4.3.2.1 Formalisation du diagramme de classes fonctionnel

La dérivation du modèle fonctionnel est inspirée des approches de transformation existantes d'UML vers B présentées dans la section 4.2.2. Elle permet de créer la machine *Functional,* qui formalise les différents éléments du diagramme de classes, comme par exemple : les classes, les attributs, les associations et les opérations.

- Les classes sont formalisées par des ensembles abstraits (abstract sets) qui permettent de représenter une abstraction d'un ensemble d'objets d'un système en B. L'ensemble des instances effectives d'une classe est représenté comme une variable dans la machine *Functional*.

[10] *http://www.clearsy.com/nos-outils/atelier-b/*

- Les attributs sont formalisés par une relation fonctionnelle associant l'ensemble des instances effectives et le type de l'attribut. Le type de cette fonction (injectif,…) dépend de la nature de l'attribut (optionnel, obligatoire, unique ou non unique).
- Les associations sont également représentées par une relation qui fait le lien entre les deux ensembles représentant les instances effectives de deux classes reliées par l'association. Le type de cette relation dépend des multiplicités de l'association.
- Les opérations de classe sont représentées par des opérations B qui définissent la partie dynamique de la machine *Functional*. Les opérations de base telles que les constructeurs/destructeurs d'instances et de liens entre instances, les getters/setters d'attributs et les getters de liens sont générées automatiquement par l'outil B4MSecure. On peut également introduire de nouvelles opérations manuellement.

Ainsi, l'outil B4MSecure permet de générer des invariants qui précisent automatiquement le type des différents éléments de classes. De ce fait, les opérations B doivent respecter les différents invariants de typage. Les différentes contraintes fonctionnelles comme les contraintes d'intégrité peuvent être également ajoutées aux spécifications B.

Dans le cadre de notre étude de cas du système d'organisation de réunions, la figure 4.6 donne un extrait de la machine *Functional* issue du diagramme de classes de la figure 2.3 (Chapitre 2). La spécification B décrit les ensembles (SETS), les variables abstraites (ABSTRACT_VARIABLES), les invariants (INVARIANT), les initialisations (INITIALISATION) et les opérations (OPERATIONS).

Les classes *Meeting*, *Invitation*, *Person* et *Change* de la figure 2.3 sont transformées respectivement en ensembles MEETING, INVITATION, PERSON et CHANGE déclarés dans la clause SETS de la machine *Functional* de la figure 4.6. Ces ensembles représentent les instances possibles de ces classes.

Les variables abstraites représentent les instances existantes des classes (ligne 10 dans la figure 4.6), les associations entre les classes (ligne 11) et les attributs de différentes classes (lignes de 12 à 15).

Les invariants de la ligne 17 montrent l'inclusion des ensembles d'instances existantes des classes comme par exemple *Meeting* dans les ensembles d'instances possibles des classes comme par exemple *MEETING*.

```
1- MACHINE
2  Functional
3- SETS
4  STR={ }
5- ;CHANGE={ }
6  ;INVITATION={ }
7  ;MEETING={ }
8  ;PERSON={ }
9  ABSTRACT_VARIABLES
10- Change, Person, Invitation, Meeting
11 , meeting_invitation, meeting_person,invitation_change, invitation_person
12- , Change__Date, Change__Time, Change__Answer
13 , Person__Name
14 , Invitation__Date, Invitation__Answer, Invitation__Confirmation
15 , Meeting__Date, Meeting__Place, Meeting__Time
16 INVARIANT
17 Change ⊆ CHANGE ∧ Person ⊆ PERSON ∧ Invitation ⊆ INVITATION ∧ Meeting ⊆ MEETING
18 ∧ meeting_invitation ∈ Invitation ⤖ Meeting ∧ meeting_person ∈ Meeting ⤖ Person
19- ∧ invitation_change ∈ Change ⤖ Invitation ∧ invitation_person ∈ Invitation ⤖ Person
20 ∧ Change__Date ∈ Change → STR ∧ ···
21 ∧ Person__Name ∈ Person ⤖ STR ∧ ···
22 ∧ Invitation__Date ∈ Invitation → STR ∧ ···
23 ∧ Meeting__Date ∈ Meeting → STR ∧ ···
24 ∧ ···
25 INITIALISATION
26 . . .
27 OPERATIONS
28- Meeting_NEW(Instance,Meeting__personValue,Meeting__DateValue,Meeting__PlaceValue,
29     Meeting__TimeValue)=
30- PRE
31- Instance ∈ MEETING ∧ Instance ∉ Meeting ∧ Meeting__personValue ∈ Person ∧
32 Meeting__DateValue ∈ STR    ∧
33- Meeting__PlaceValue ∈ STR   ∧
34 ∧ Meeting__TimeValue ∈ STR
35 THEN     Meeting := Meeting ∪ {Instance}
36 || meeting_person:= meeting_person ∪ {(Instance↦Meeting__personValue)}
37 || Meeting__Date:= Meeting__Date ∪ {(Instance↦Meeting__DateValue)}
38- || Meeting__Place:= Meeting__Place ∪ {(Instance↦Meeting__PlaceValue)}
39 || Meeting__Time:= Meeting__Time ∪ {(Instance↦Meeting__TimeValue)}
40 END;
41 result ← Meeting__getDate(Instance)=
42 PRE
43 Instance ∈ Meeting ∧ Instance ∈ dom(Meeting__Date)
44 THEN     result := Meeting__Date(Instance)
45- END;
46 . . .
47 END
```

Figure 4.6. La machine Functional

Les invariants des lignes 18 et 19 sont relatifs aux associations des différentes classes et permettent de spécifier leurs multiplicités. Les invariants des lignes 20 à 24 spécifient les types des attributs des différentes classes.

Pour la partie dynamique de la machine *Functional*, *(i)* on peut initialiser les différentes variables abstraites dans la ligne 26, *(ii)* les opérations de classes sont représentées par des opérations B. Les lignes 28 à 40 présentent un exemple de constructeur des instances de la classe *Meeting* avec toutes les préconditions qui assurent la cohérence de la machine *Functional.* Les lignes 41 à 45 présentent un exemple de getter de l'attribut *Date* de la classe *Meeting.*

Dans la section suivante, la machine *Functional* sera utilisée comme base pour la spécification du filtre formel RBAC.

4.3.2.2 Formalisation de modèles SecureUML

A partir des modèles SecureUML et de la machine *Functional*, l'outil B4MSecure génère automatiquement le filtre formel RBAC (la machine RBAC). Ce filtre permet de contrôler l'accès aux opérations de la machine *Functional* qui représentent les opérations des classes fonctionnelles. Cela se fait en associant à chaque opération de la machine *Functional*, une opération sécurisée dans la machine de sécurité *RBAC*. Quand un utilisateur souhaite exécuter une opération fonctionnelle, il se connecte au moyen d'une session à un ou plusieurs rôles, ce qui permet de calculer l'ensemble des opérations autorisées pour cet utilisateur. L'opération sécurisée dans la machine *RBAC* permet ensuite de vérifier si l'utilisateur courant est autorisé à appeler l'opération fonctionnelle correspondante. Le filtre RBAC est défini par deux machines formelles : *UserAssignments* et *RBAC*

4.3.2.2.1 La machine UserAssignments

La figure 4.7 présente un extrait de la machine *UserAssignments*. Cette machine spécifie l'affectation des utilisateurs aux rôles. Elle définit l'ensemble des utilisateurs du système (ligne 4) et l'ensemble des rôles (ligne 5) ainsi que les variables *roleOf*, *Roles_Hierarchy*, *currentUser*, *SSD_mutex*, *DSD_mutex et Session* (ligne 7) dont l'explication est donnée ci-dessous :

- La relation *roleOf* (ligne 10) spécifie l'affectation des utilisateurs aux rôles en associant à chaque utilisateur un ensemble de rôles. La ligne 16 initialise cette relation.
- La relation *Roles_Hierarchy* (ligne 9) formalise la hiérarchie de rôles.
- L'invariant *closure1(Roles_Hierarchy) /\ id(ROLES) = {}* (ligne 10) indique que la hiérarchie de rôles ne doit pas contenir de cycles.
- La variable *currentUser* (ligne 11) sert pour l'animation du modèle et permet d'identifier l'utilisateur courant du système.
- La relation *session* (ligne 12) formalise la connexion des utilisateurs aux rôles.
- Les contraintes de *SoD* sont aussi spécifiées par des invariants.
- La clause *OPERATIONS* spécifie un ensemble d'opérations qui permettent de gérer les affectations et les connexions des utilisateurs aux rôles. Les lignes 19 à 24 présentent un exemple de l'opération *Connect* qui connecte un utilisateur au système à travers une session comprenant un sous-ensemble de ses rôles.

```
1  MACHINE
2  UserAssignments
3  SETS
4  USERS = {Paul,Bob,Jack}
5  ; ROLES = {Participant,Initiator}
6  VARIABLES
7  roleOf,  Roles_Hierarchy, currentUser, SSD_mutex, DSD_mutex, Session
8  INVARIANT
9  Roles_Hierarchy ∈ ROLES ↔ ROLES ∧
10 roleOf ∈ USERS → ℙ(ROLES) ∧  closure1(Roles_Hierarchy) ∩ id(ROLES) = ∅ ∧
11 currentUser ∈ USERS ∧
12 Session ∈ USERS↔ ROLES ∧ ∀(uu).(uu∈USERS ∧ uu∈dom(Session)⇒ Session[{uu}]⊆ roleOf(uu))
13 ∧ SSD_mutex  ...
14 ∧ DSD_mutex  ...
15 INITIALISATION
16 roleOf := {(Paul↦{Participant}),(Bob↦{Initiator,Participant}),(Jack↦{Initiator})}
17 ||....
18 OPERATIONS
19 connect(user,roleSet) =
20 PRE
21 user ∈ USERS ∧ user∉dom(Session) ∧ roleSet∈ℙ1(ROLES) ∧ roleSet⊆roleOf(user)
22 THEN
23 Session := SessionU({user}*roleSet)
24 END;
25 . . .
26 END
```

Figure 4.7. La machine UserAssignments

4.3.2.2.2 La machine RBAC

La figure 4.8 présente un extrait de la machine *RBAC*. Cette machine inclut la machine *UserAssignments et* la machine *Functional*. Elle consiste à formaliser les permissions des différents rôles aux éléments des classes fonctionnelles protégées. Cela permet de trouver l'ensemble des opérations de classes permises à un utilisateur selon ses rôles. Pour pouvoir exprimer les permissions, la structure du diagramme de classes fonctionnel a été intégrée dans la machine *RBAC*. Les ensembles *ENTITIES* (classes), *Attributes* (attributs), *Operations* (opérations) et *KindsOfAtt* (visibilité), définis dans les lignes 7 à 10 de la figure 4.8, contiennent les éléments du diagramme de classes fonctionnel. Les relations entre ces ensembles sont :

- *AttributeKind* : indique si un attribut de classe est privé ou public.
- *AttributeOf* : indique la classe dans laquelle un attribut est encapsulé.
- *OperationOf, constructorOf et destructorOf* : indiquent la classe dans laquelle une opération est encapsulée.
- *setterOf, getterOf* : rattachent les setters et les getters à leurs attributs.

En effet, le calcul des permissions dépend des types d'opération (*OperationOf, constructorOf,destructorOf, setterOf et getterOf*).

La machine *RBAC* ajoute des éléments relatifs aux permissions :

- L'ensemble *PERMISSIONS* (ligne 11) spécifie les différentes permissions définies dans les modèles SecureUML.

- L'ensemble *ActionsType* (ligne 12) définit les différents types d'action (lecture, écriture, etc.) des permissions SecureUML.
- *PermissionAssignement* (ligne 23) est une fonction totale de l'ensemble PERMISSIONS vers le produit cartésien (ROLES * ENTITIES).
- La relation *EntityActions* (ligne 24) représente les permissions exprimées sur une entité de manière globale à travers les actions définies dans *ActionsType*.
- La relation *MethodActions* (ligne 25) définit les permissions spécifiques à certaines opérations de la classe cible de permission SecureUML.
- La relation *isPermitted* (ligne 26) spécifie les opérations de classes autorisées aux rôles du système. Elle est calculée à partir des permissions SecureUML.

```
1  MACHINE
2  RBAC
3  INCLUDES
4  Functional,
5  UserAssignments
6  SETS
7  ENTITIES = {Meeting_Label,Change_Label,Person_Label,Invitation_Label};
8  Attributes = {Meeting_Date_Label,…};
9  Operations = { Meeting_NEW_Label ,Meeting__getDate_Label,…};
10 KindsOfAtt = {public, private};
11 PERMISSIONS = {InitiatorMeeting, ParticipantMeeting,…};
12 ActionsType = {read, create, modify, delete, privateRead, privateModify,fullAccess};
13 Stereotypes = {readOp, modifyOp};
14 VARIABLES
15 AttributeKind, AttributeOf, OperationOf, constructorOf, destructorOf, setterOf, getterOf,
16 PermissionAssignement,  EntityActions, MethodActions, StereotypeOps, isPermitted
17 INVARIANT
18 AttributeKind ∈ Attributes → KindsOfAtt ∧ AttributeOf ∈ Attributes → ENTITIES ∧
19 OperationOf ∈ Operations → ENTITIES ∧ constructorOf ∈ Operations ⤖ ENTITIES ∧
20 destructorOf ∈ Operations ⤖ ENTITIES ∧ setterOf ∈ Operations ⤖ Attributes ∧
21 getterOf ∈ Operations ⤖ Attributes ∧ StereotypeOps ∈  Stereotypes ↔ Operations ∧
22 setterOf ∩ getterOf = ∅ ∧
23 PermissionAssignement ∈ PERMISSIONS → (ROLES * ENTITIES) ∧
24 EntityActions ∈ PERMISSIONS ⇸ ℙ(ActionsType) ∧
25 MethodActions ∈ PERMISSIONS ⇸ ℙ(Operations) ∧
26 isPermitted ∈ ROLES ↔ Operations
27 INITIALISATION
28 isPermitted := ∅||…
29 OPERATIONS
30 secure_Meeting_NEW(Instance,Meeting__personValue,Meeting__DateValue,Meeting__PlaceValue,
31     Meeting__TimeValue)= PRE…
32 THEN     SELECT
33 Meeting_NEW_Label ∈ isPermitted[currentRole]
34 THEN  Meeting_NEW(Instance,Meeting__personValue,Meeting__DateValue,Meeting__PlaceValue,
35         Meeting__TimeValue)
36 END
37 END;
38 ...
39 END
```

Figure 4.8. La machine RBAC

L'initialisation des variables *PermissionAssignment*, *EntityActions* et *MethodActions*, générée automatiquement par l'outil B4MSecure, permet de créer le filtre RBAC. La relation *isPermitted*, initialisée à l'ensemble vide, est ensuite complétée par tous les couples (rôle, opération) permis par la politique RBAC après la connexion des utilisateurs aux rôles dans le cadre d'une session.

Dans la clause OPERATIONS de la machine *RBAC*, chaque opération de la machine *Functional* est associée à une opération sécurisée pour vérifier si l'utilisateur courant (selon ses rôles) possède la permission d'effectuer cette opération. Une garde de la forme *operation : isPermitted[currentRole]* est utilisée pour limiter l'accès à une opération fonctionnelle. *currentRole* correspond à l'ensemble des rôles activés par un utilisateur dans une session. Les lignes 30 à 37 de la figure 4.8 présentent un exemple d'une opération sécurisée correspondant au constructeur d'instances de la classe *Meeting.*

Les contraintes d'autorisation associées aux permissions SecureUML peuvent être également traduites de l'OCL vers B et exprimées comme des gardes des opérations sécurisées. La version actuelle de la plateforme B4MSecure ne supporte pas la traduction automatique des contraintes d'autorisation. Il faut donc les injecter manuellement.

4.3.3 Formalisation de modèles BAAC@UML

Cette section présente notre démarche de spécification formelle d'une politique RBAC au niveau des activités d'un BP. La spécification de la vue dynamique d'une politique RBAC en B permet la mise en place de la politique de sécurité au niveau de chaque activité métier, en ne donnant le droit d'accéder aux actions de l'activité qu'aux utilisateurs autorisés par la politique RBAC.

Chaque activité est formalisée par une seule machine formelle (*FormalActivity*). La machine *FormalActivity* inclut via un opérateur INCLUDES la machine *RBAC* pour tenir compte de la politique de sécurité exprimée au niveau statique (voir figure 4.5). Les machines d'activités incluent également la machine *Flow* qui définit des éléments de représentation de flot de contrôle de d'activité. L'affectation des rôles aux activités est formalisée par la machine *ActivityAssignmentToRoles*. Dans cette section, nous détaillons les trois machines : *ActivityAssignmentToRoles, Flow et FormalActivity.*

4.3.3.1 La machine ActivityAssignmentToRoles

La figure 4.9 présente la machine *ActivityAssignmentToRoles* qui formalise l'affectation des rôles aux activités. Cette machine voit la machine *UserAssignments* issue des modèles SecureUML. Les activités sont spécifiées par l'ensemble *ACT* (lignes 6 à 9). Les rôles sont définis par l'ensemble *Role* dans la machine *UserAssignments*. La relation fonctionnelle *roleOfActivity* (ligne 13) spécifie les liens entre *ACT* et *Role* en associant à chaque activité un ensemble de rôles.

Les deux opérations *assignRoleToActivity* (lignes 21 à 27) et *unassignRoleOfActivity* (lignes 28 à 34) permettent respectivement d'affecter et de désaffecter une activité à un rôle. L'initialisation (lignes 15 à 19) spécifie l'affectation des rôles aux différentes activités du système d'organisation de réunions.

```
1- MACHINE
2      ActivityAssignmentToRoles
3- SEES
4       UserAssignments
5- SETS
6-  ACT={FollowAnswer,
7     InviteParticipant,
8-    ConsultMeetingMembers,CreateMeeting,
9     ReplyToInvitation,ViewConfirmation}
10- VARIABLES
11     roleOfActivity
12- INVARIANT
13     roleOfActivity∈ ACT↠ℙ(ROLES)
14- INITIALISATION
15-    roleOfActivity:={(FollowAnswer↦{Initiator}),
16    (ConsultMeetingMembers↦{Participant,Initiator}),
17    (CreateMeeting↦{Initiator}),(InviteParticipant↦{Initiator}),
18    (ReplyToInvitation↦{Participant}),
19   (ViewConfirmation↦{Participant})}
20- OPERATIONS
21 assignRoleToActivity(act,role) =
22-   PRE
23     act ∈ ACT ∧ role∈ROLES ∧
24     role∉(roleOfActivity(act))
25-   THEN
26          roleOfActivity := ({act}◀roleOfActivity) ∪
27          {(act↦ ({role} ∪ roleOfActivity(act)) ) } END;
28 unassignRoleOfActivity(act,role) =
29-   PRE
30      act ∈ ACT ∧ role∈ROLES ∧ role∈roleOfActivity(act)
31      ∧ role∉ran(Session)
32-   THEN
33      roleOfActivity := ({act}◀roleOfActivity) ∪
34      {(act↦ (roleOfActivity(act)-{role}) ) }END
35 END
```

Figure 4.9. La machine ActivityAssignmentToRoles

4.3.3.2 La machine Flow

Afin de contrôler le flux d'actions et de tâches de l'activité et d'obtenir des traces d'exécution, nous définissons la machine *Flow* de la figure 4.10. Dans cette machine une trace d'exécution est une séquence de flots d'un diagramme d'activités, définie par l'ensemble énuméré *Ctrl Flow* (ligne 4).

L'opération *addTrace* (lignes 20 à 23) et la variable *flowTrace* (ligne 7) gèrent la trace d'exécution de l'activité et permettent de sauvegarder la trajectoire suivie dans chaque exécution. Pour le modèle BAAC@UML de la figure 3.12 (Chapitre 3), il

existe quatre trajectoires possibles qui correspondent aux quatre scénarios de l'activité.

La variable *currentFlow* (ligne 6) et l'opération *setCurrentFlow* (lignes 15 à 19) permettent de gérer l'enchaînement des actions et des tâches de l'activité. Par exemple, le diagramme d'activités de la figure 3.12 montre que la tâche "*confirm invitation*" ne peut être exécutée que si le flot courant contient le tag *cf5*.

La ligne 10 spécifie des contraintes sur le flot de contrôle de l'activité sous forme d'invariants qui doivent être respectés par l'opération *setCurrentFlow*.

```
1- MACHINE
2     Flow
3- SETS
4      CtrlFlow = {cf1, cf2 ,cf3,cf4,cf5...}
5- VARIABLES
6      currentFlow,
7      flowTrace
8- INVARIANT
9    currentFlow ⊆ CtrlFlow ∧ flowTrace ∈ seq(CtrlFlow)
10   ∧ (currentFlow={cf1} ∨ currentFlow={cf5,cf6,cf7}...
11- INITIALISATION
12     currentFlow := ∅
13     || flowTrace := ∅
14- OPERATIONS
15 setCurrentFlow(flow) =
16     PRE flow ⊆ CtrlFlow ∧ (flow ={cf1} ∨ flow={cf5,cf6,cf7}...
17-    THEN
18         currentFlow := flow
19     END;
20 addTrace(flow) =
21-    PRE flow ∈ CtrlFlow THEN
22         flowTrace := flowTrace ← flow
23     END
24 END
```

Figure 4.10. La machine Flow

4.3.3.3 La machine FormalActivity

Chaque activité BAAC@UML est spécifiée par une machine B (FormalActivity) qui considère la spécification formelle de modèles SecureUML (la machine *RBAC*) et la machine *Flow* présentée dans la section précédente.

La machine *FormalActivity* formalise les éléments fonctionnels (tels que Activité, Tâche, Action concrète et Flot de contrôle) et les éléments de sécurité (tels que Rôle et Précondition de contrôle d'accès) du modèle BAAC@UML.

Nous proposons deux approches pour la spécification de la machine d'activité :

- La première approche est basée sur les actions (ABAM ou Action-Based Activity Machine). Elle est adaptée à des activités de taille moyenne (moins de 20 actions concrètes).
- La deuxième approche est basée sur les tâches (TBAM ou Task-Based Activity Machine). Elle est adaptée à des activités de grande taille (au-delà de 20 actions concrètes).

4.3.3.3.1 ABAM

La figure 4.11 présente un exemple de la traduction de l'activité *FollowAnswer* de la figure 3.12 en machine B selon la première approche (ABAM). Dans cette machine, chaque action concrète est représentée par une opération. Cette opération fait appel à l'opération sécurisée correspondante dans la machine *RBAC*. L'action concrète *M.getDate* du modèle BAAC@UML de la figure 3.12, par exemple, est représentée par une opération qui fait appel à l'opération sécurisée *secure_Meeting_getDate* de la machine *RBAC*. Cela permet d'exprimer la protection du filtre *RBAC* au niveau des machines d'activités.

Dans la quatrième ligne de la machine d'activité *ABAM*, les variables *Mx et Ix* formalisent les paramètres de l'activité *FollowAnswer* (les objets invoqués par les actions concrètes). Quand une activité est commencée, ces variables doivent être instanciées par des objets pertinents. La variable booléenne *opened* déclare l'ouverture de l'activité. La variable *Result* reçoit les résultats des opérations de lecture.

Les invariants de la machine d'activité *ABAM* (lignes 6 à 8) spécifient le typage des variables et la précondition d'activité (stéréotypée par *assignedRoles* dans le modèle BAAC@UML) qui doit être satisfaite par l'activité. Dans notre exemple, *assignedRoles* signifie que seul un initiateur peut exécuter l'ensemble de l'activité. Nous considérons que, une fois une session ouverte et un utilisateur connecté, le rôle *Initiator* est l'un des rôles de l'utilisateur, et le rôle *Initiator* est activé. Il faut noter que si l'activité est affectée à plusieurs rôles, l'utilisateur peut se connecter et activer un ou une partie de ces rôles.

Les préconditions locales de contrôle d'accès sont traduites du langage OCL vers B et spécifiées dans la clause *DEFINITIONS*. Elles consistent à préconditionner certaines opérations dans la machine d'activité. Ce sont les opérations qui correspondent aux actions concrètes rattachées à des *LACPreCondition* dans les modèles BAAC@UML. *FA-MeetingReadUpdate* (ligne 13) est une expression booléenne qui retourne *vrai* lorsque l'utilisateur courant est le créateur de l'objet *Mx* de la classe *Meeting*. *FA-InvitationFullAccess* (ligne 14) vérifie si l'utilisateur courant

est le créateur de l'instance du meeting associée à une instance *Ix* de la classe *Invitation*.

```
1  MACHINE
2  ABAM_FollowAnswer
3  INCLUDES RBAC, Flow
4  VARIABLES Result, Mx, Ix, opened
5  INVARIANT
6  Result ∈ STR ∧ Mx ∈ Meeting ∧ Ix ∈ INIVITATION ∧ opened ∈ BOOL
7  ∧ (opened = TRUE ⇒ Initiator ∈ roleOf(currentUser)
8                     ∧ {Initiator} ⊆ ran(Session))
9  INITIALISATION
10 opened := FALSE || Result := nones || Mx :∈ Meeting
11 || Ix :∈ Invitation
12 DEFINITIONS
13 FA_MeetingReadUpdate == meeting_person(Mx) = currentUser;
14 FA_InvitationFullAccess == meeting_person(meeting_invitation(Ix))= currentUser;
15 OPERATIONS
16 Open_activity (actUser, Mi, Ii) =
17   PRE
18     actUser ∈ USERS ∧ Initiator ∈ roleOf(actUser)
19    ∧ Mi ∈ Meeting ∧ Ii ∈ Invitation  ∧ meeting_invitation(Ii) = Mi
20   THEN
21    setPermissions;
22    Connect(actUser,{Initiator});
23    changeCurrentUser(actUser);
24    Mx := Mi ; Ix := Ii ;
25    opened := TRUE ;
26    setCurrentFlow({cf1})
27   END ;
28 M.getDate= PRE FA_MeetingReadUpdate ∧  cf1 ∈ currentFlow
29           THEN Result ← secure_Meeting__getDate(Mx);
30                addTrace(cf1); setCurrentFlow({cf2}) END;
31 M.getTime= PRE FA_MeetingReadUpdate ∧  cf2 ∈ currentFlow
32           THEN Result ← secure_Meeting__getTime(Mx);
33                addTrace(cf2); setCurrentFlow({cf3}) END;
34 M.getPlace= PRE FA_MeetingReadUpdate ∧  cf3 ∈ currentFlow
35           THEN Result ← secure_Meeting__getPlace(Mx);
36                addTrace(cf3); setCurrentFlow({cf4}) END;
37 I.getAnswer= PRE FA_InvitationFullAccess ∧  cf4 ∈ currentFlow
38           THEN Result ← secure_Invitation__getAnswer(Ix);
39                addTrace(cf4); setCurrentFlow({cf5, cf6, cf7}) END;
40 ...
41 END
```

Figure 4.11. La machine d'activité ABAM

L'opération *Open_activity* (lignes 16 à 27) spécifie l'ouverture d'une session par l'utilisateur avant de commencer l'exécution de l'activité. Afin de commencer une activité, il faut connecter le bon utilisateur et utiliser des objets pertinents pour les classes *Meeting* et *Invitation*. L'opération *Open_activity* connecte un utilisateur avec le rôle *Initiator* et active son rôle dans la session correspondante, puis elle déclare l'ouverture de l'activité et initie le flot de contrôle à *cf1*.

Les autres opérations de machine d'activité *ABAM* formalisent le comportement de l'activité en spécifiant chaque action concrète par une seule opération B. Pour chaque

action concrète, nous prenons en compte: le flot de contrôle permettant de démarrer l'action, l'éventuelle précondition de contrôle d'accès associée à l'action, la gestion de la trace du flux de contrôle de l'activité, et enfin le flot de contrôle permettant de démarrer l'action suivante. La figue 4.11 donne des exemples des opérations B spécifiant les actions concrètes *M.getDate* (lignes 28 à 30), *M.getTime* (lignes 31 à 33), *M.getPlace* (lignes 34 à 36) et *I.getAnswer* (lignes 37 à 39).

4.3.3.3.2 TBAM

Pour la machine d'activité TBAM, les opérations B représentent des tâches plutôt que des actions concrètes (le cas de la machine d'activité ABAM). Nous illustrons cette machine à travers l'exemple de modèle BAAC@UML de la figure 4.12. Ce modèle spécifie des contraintes supplémentaires pour forcer le contrôle d'accès à l'activité *Follow answer*.

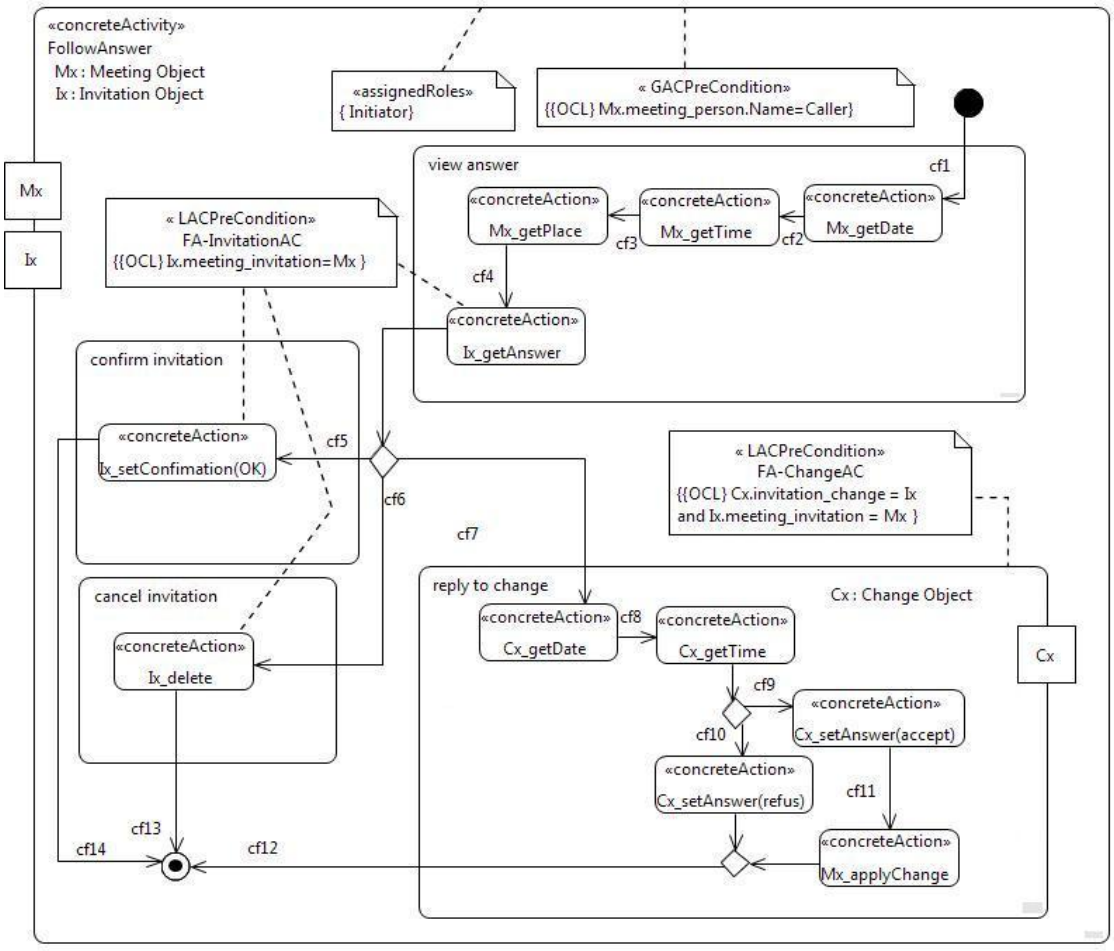

Figure 4.12. Modèle BAAC@UML de contrôle d'accès à l'activité concrète Follow answer v2

Le modèle BAAC@UML spécifie une précondition contextuelle globale de contrôle d'accès qui doit être vérifiée dans l'exécution de l'ensemble de l'activité. Elle est exprimée avec une précondition d'activité stéréotypée comme *<<GACPreCondition >>* en utilisant le langage OCL. Dans notre exemple, cela garantit que l'utilisateur exécutant l'activité est le créateur de la réunion (l'instance *Mx* de la classe *Meeting*).

Le modèle BAAC@UML spécifie également la précondition locale de contrôle d'accès *FA_InvitationAC* qui vérifie que le créateur de la réunion ne peut lire, modifier ou supprimer que les invitations de sa réunion (l'instance *Ix* de la classe *Invitation* associée à *Mx*).

La précondition locale de contrôle d'accès *FA_ChangeAC* associée à la tâche *reply_to_change* vérifie que le créateur de la réunion ne peut répondre qu'aux changements associés aux invitations de sa réunion (l'instance *Cx* de la classe *Change* associée à *Ix*).

La partie statique de la spécification *B* issue du modèle BAAC@UML de la figure 4.12 est représentée dans la figure 4.13. Dans cette partie de la machine d'activité *TBAM*, nous avons retenu les mêmes principes que ceux de la machine *ABAM* pour la formalisation des paramètres de l'activité et de la précondition *assignedRoles*. La précondition globale de contrôle d'accès (GACPreCondition) est traduite de l'OCL vers B et exprimée comme invariant de la machine (ligne 9). Elle retourne vrai lorsque l'utilisateur courant est le créateur de l'instance *Mx* de la classe *Meeting*. La précondition locale *FA_InvitationAC* est également transformée en B et spécifiée dans la clause DEFINITIONS (ligne 14). Elle vérifie que l'invitation courante (*Ix*) correspond à la réunion courante (*Mx*). Cette condition est vérifiée avant l'exécution des actions *Ix_ getAnswer*, *Ix_ setConfirmation* et *Ix_ delete*.

```
1- MACHINE
2  TBAM_FollowAnswer
3  INCLUDES RBAC, Flow
4  VARIABLES Result, Mx, Ix, opened
5- INVARIANT
6  Result ∈ STR ∧ Mx ∈ Meeting ∧ Ix ∈ INIVITATION ∧ opened ∈ BOOL
7- ∧ (opened = TRUE ⇒ Initiator ∈ roleOf(currentUser)
8                    ∧ {Initiator} ⊆ ran(Session)
9                     ∧ meeting_person_1(Mx) = currentUser)
10- INITIALISATION
11 opened := FALSE || Result := nones || Mx :∈ Meeting
12 || Ix :∈ Invitation
13- DEFINITIONS
14 FA_InvitationAC == meeting_invitation(Ix) = Mx;
```

Figure 4.13 : La partie statique de la machine d'activité TBAM

La partie dynamique de la machine d'activité *TBAM* issue du modèle BAAC@UML de la figure 4.12 est représentée dans la figure 4.14. L'opération *Open_activity* de la machine d'activité *ABAM* est retenue. Nous avons en plus exprimé *GACPreCondition* et *FA_InvitationAC* comme précondition de cette opération pour éviter toute violation des invariants de la machine.

Les autres opérations de notre machine B spécifient le comportement de l'activité en utilisant une seule opération par tâche. Pour chaque tâche, nous considérons le flot de contrôle permettant de démarrer la tâche, les différentes préconditions locales de contrôle d'accès, la spécification de la trace du flux de contrôle, et le flot de contrôle permettant de démarrer la tâche suivante.

La partie X de la figure 4.14 présente l'opération B qui formalise la tâche *view_answer* du modèle BAAC@UML de la figure 4.12. Cette tâche est spécifiée par une séquence de substitutions qui fait appel à des actions concrètes (représentées par les opérations sécurisées correspondantes de la machine *RBAC*) suivies par les opérations *addTrace* qui gèrent la trace du flux de contrôle de l'activité. La dernière action *secure_ Invitation__ getAnswer*, pré-conditionnée par *FA_ InvitationAC*, clôture la tâche et par conséquent, elle est suivie par l'opération *setCurrentFlow ({cf5,cf6, cf7})* qui active les tâches suivantes.

X

```
view_answer =
PRE cf1 ∈ currentFlow THEN
  Result ← secure_Meeting__getDate(Mx) ; addTrace(cf1);
  Result ← secure_Meeting__getTime(Mx) ; addTrace(cf2);
  Result ← secure_Meeting__getPlace(Mx) ; addTrace(cf3);
  PRE  FA_InvitationAC THEN
    Result ← secure_Invitation__getAnswer(Ix) ; addTrace(cf4) ;
    setCurrentFlow({cf5, cf6, cf7})
  END
END;
```

Y

```
reply_to_change(Cx) =
   PRE cf7 ∈ currentFlow ∧ Cx ∈ Change
       ∧ invitation_change(Cx)=Ix ∧ meeting_invitation(Ix)= Mx
   THEN
      Result ← secure_Change__getDate(Cx); addTrace(cf7) ;
      Result ← secure_Change__getTime(Cx); addTrace(cf8);
   CHOICE
      secure_Change__setAnswer(Cx,accept); addTrace(cf9);
      secure_Meeting__applyChange(Mx) ; addTrace(cf11) ;
      setCurrentFlow({cf12})
   OR
      secure_Change__setAnswer(Cx,refus) ; addTrace(cf10) ;
      setCurrentFlow({cf12})
   END
END;
```

Figure 4.14. La partie dynamique de la machine d'activité TBAM

La partie Y de la figure 4.14 donne la spécification B de la tâche *reply_ to_ change*. Le paramètre *Cx* de la tâche représente une instance de la classe *Change*. La tâche commence lorsque le flot de contrôle courant contient *cf7* et la précondition locale *FA_ChangeAC* est satisfaite. *FA_ChangeAC* est traduite de l'OCL (voir figure 4.12) vers B. Dans cette description, nous avons deux choix en fonction des flots de contrôle *cf9* ou *cf10*. Le choix est non déterministe et dépend de l'initiateur qui peut sélectionner une voie ou l'autre. La trace produite par cette opération pourrait être la séquence *[cf7, cf8, cf9, cf11]* ou la séquence *[cf7, cf8, cf10]* en fonction du chemin choisi par l'initiateur.

4.4 Validation formelle d'une politique RBAC

La validation des modèles est devenue une préoccupation majeure dans le cadre de l'approche IDM. Cette validation est autant plus nécessaire lorsque la fonction du système est critique. L'objectif de la traduction des modèles BAAC@UML en spécifications B est de vérifier rigoureusement la politique RBAC exprimée au niveau des activités d'un BP en utilisant les techniques de V&V de la méthode B présentées dans la section 4.2.3, telles que l'animation et la preuve.

Notre profil BAAC@UML introduit les concepts de *rôle* et *précondition de contrôle d'accès* pour garder l'accès des utilisateurs aux différentes activités d'un BP. Ainsi, les modèles BAAC@UML dépendent du diagramme de classes fonctionnel (les actions concrètes font appel aux opérations de classes et les préconditions de contrôle d'accès font référence aux classes et à leurs associations) et des modèles SecureUML (Les préconditions de contrôle d'accès se réfèrent aux contraintes d'autorisation). La cohérence entre ces modèles doit être étudiée pour éviter toute contradiction.

Dans les chapitres 2 et 3, nous avons défini un méta-modèle qui spécifie la structure de nos extensions des diagrammes d'activités et la sémantique statique de leurs liens avec les modèles SecureUML et le diagramme de classes fonctionnel. Nous avons également défini un ensemble de règles OCL pour faire respecter certaines propriétés de cohérence entre ces modèles. Notre méta-modèle et les règles OCL qui lui sont associées garantissent que:

(P1) Les actions concrètes du modèle BAAC@UML font appel aux opérations du diagramme de classes fonctionnel.

(P2) Les rôles associés au modèle d'activité BAAC@UML peuvent exécuter toutes les actions des différentes tâches de l'activité. Les modèles SecureUML ne bloquent aucun de ces rôles.

La formalisation des modèles BAAC @ UML en B permet de vérifier d'autres propriétés (cas d'incohérence). Nous cherchons à assurer que:

(P3) Les modèles SecureUML autorisent un utilisateur, connecté aux rôles d'une activité BAAC@UML et remplissant les préconditions de contrôle d'accès, à exécuter les différents scénarios de l'activité.
(P4) Les tâches de l'activité et leurs actions préservent *GACPrecondition* et *assignedRoles*.
(P5) Les préconditions de contrôle d'accès peuvent être évaluées à *vrai* ou *faux* en fonction de l'état du système, de l'utilisateur et de ses rôles.

Les prouveurs, comme Atelier B, produisent des obligations de preuve et permettent de vérifier que les contraintes de sécurité sont conservées dans l'exécution des tâches de l'activité. Ceci correspond à la preuve de correction de la propriété P4.

L'animation, en utilisant ProB par exemple, permet de simuler l'exécution des scénarios d'activité par un utilisateur connecté à des rôles (propriété de P3). Dans l'animation, les préconditions de contrôle d'accès seront évaluées pour permettre l'exécution des tâches de l'activité (propriété P5). L'animation vérifie également que les modèles BAAC@UML spécifient les besoins de l'analyste, et que la politique RBAC est bien respectée au niveau des modèles.

Les sections 4.4.1 et 4.4.2 discutent respectivement de la preuve et de l'animation des spécifications B issues des modèles BAAC@UML.

4.4.1 Preuve de spécifications B

Atelier B[11] (ClearSy, 2014a) est un outil industriel, développé par la société *ClearSy*, qui permet une utilisation opérationnelle de la méthode formelle B pour le développement de logiciels prouvés. Il dispose d'un *prouveur automatique* pour la démonstration de la plupart des obligations de preuve et d'un *prouveur interactif* pour la détection des erreurs et la finalisation de la preuve. Les obligations de preuve sont expliquées dans la section 4.2.3.1, elles représentent un ensemble de théorèmes qui servent à prouver certaines propriétés des machines B, comme les propriétés d'invariance.

Avant la génération des obligations de preuve, nous avons d'abord commencé par l'analyse syntaxique des machines B générées à partir de modèles BAAC@UML, de modèles SecureUML et du diagramme de classes fonctionnel. En effet, ce contrôle permet de vérifier que les différentes machines abstraites respectent les règles de

[11] *http://www.atelierb.eu/*

construction du langage B. Il est nécessaire pour permettre la production des obligations de preuve.

Après la correction syntaxique, nous avons généré automatiquement les obligations de preuve associées aux différentes machines abstraites. Il y a des obligations de preuve pour l'initialisation et d'autres pour chacune des opérations. Elles prouvent la préservation des propriétés invariantes des machines abstraites. Par exemple:

- L'initialisation et les opérations de la machine *TBAM_FollowAnswer* (figures 4.13 et 4.14) préservent les invariants ci-dessous qui assurent les préconditions d'activité *assignedRoles* et *GACPreCondition* (propriété P4). Nous avons exprimé ces préconditions comme des invariants. Ces derniers garantissent que ces conditions seront vérifiées à l'entrée de chaque action du modèle BAAC@UML.

$$opened = TRUE \Rightarrow roleOfActivity(FollowAnswer) \in roleOf(currentUser) \\ \wedge \{ roleOfActivity(FollowAnswer) \} = ran(Session) \\ \wedge meeting\ person(Mx) = currentUser$$

- L'initialisation et les opérations de la machine *Flow* (figure 4.10) préservent les invariants ci-dessous qui assurent la correction du flux de tâches.

$$currentFlow = \emptyset \vee currentFlow=\{cf1\} \\ \vee currentFlow=\{cf5,cf6,cf7\} \vee currentFlow=\{cf12\} \\ \vee currentFlow=\{cf13\} \vee currentFlow=\{cf14\}$$

Les obligations de preuve pour la machine RBAC générée à partir de modèles SecureUML n'ont pas beaucoup d'intérêt parce que le modèle traduit est basé sur une spécification réutilisable qui a déjà été prouvée correcte par rapport à ces obligations de preuve. La machine RBAC est considérée comme un filtre qui ne propose pas de nouvelles opérations et n'ajoute pas un comportement supplémentaire. Il permet donc de limiter l'accès à certaines opérations en fonction du rôle.

4.4.2 Animation de spécifications B

ProB (Leuschel, Butler, 2008) supporte plusieurs techniques de validation telles que *l'animation* et *model-checking*. La technique d'animation est présentée dans la section 4.2.3.2. Elle simule l'évolution de l'état du système. Le premier état est calculé à partir de l'initialisation. Les états suivants dépendent des opérations qui doivent satisfaire les préconditions et les invariants. Nous tirons profit des capacités d'animation de l'outil ProB pour effectuer des tests sur les machines d'activités.

L'animation de la machine d'activité BAAC@UML simule l'exécution de scénarios d'activité, par des utilisateurs connectés à des rôles pour un état donné du modèle fonctionnel. Pour la machine d'activité ABAM, les scénarios nominaux correspondent à une succession particulière d'actions concrètes s'exécutant du début à la fin de l'activité. Pour la machine d'activité TBAM, les scénarios nominaux correspondent à une succession de tâches qui s'exécutent du début à la fin de l'activité. Le modèle BAAC@UML de la figure 4.12 inclut quatre scénarios nominaux possibles (basés sur les tâches).

(S1) Open_ activity, view_ answer, confirm_ invitation.
(S2) Open_ activity, view_ answer, cancel_ invitation.
(S3) Open_ activity, view_ answer, reply_ to_ change (En acceptant le changement).
(S4) Open_ activity, view_ answer, reply_ to_ change (En refusant le changement).

Selon les conventions de (Legeard et al., 2002), on définit deux types de tests (positif et négatif) pour vérifier les propriétés de sécurité, exprimées au niveau de la machine d'activité BAAC@UML, dans l'exécution des scénarios nominaux par un utilisateur :

- Les tests positifs vérifient les propriétés P3 et P5 définies au début de la section 4.4. Ils assurent qu'un utilisateur connecté aux rôles de l'activité et remplissant les préconditions de contrôle d'accès, peut jouer les scénarios d'activité et ne sera pas bloqué par les permissions SecureUML (propriété P3), et que les préconditions de contrôle d'accès peuvent être évaluées à vrai (propriété P5).
- Les tests négatifs vérifient que les préconditions de contrôle d'accès peuvent être évaluées à faux (propriété P5).

Les propriétés de sécurité doivent être vérifiées à un instant donné, et pour un état donné au moment de l'exécution de l'activité par un utilisateur. Afin de réaliser des animations sur une machine d'activité, il faut, d'une part, définir des instances structurelles (Objets et liens) nécessaires pour l'exécution des différentes actions concrètes de l'activité, et d'autre part, créer des utilisateurs qui peuvent exécuter l'activité avec leur affectation à des rôles.

4.4.2.1 Les utilisateurs et les instances de classes

L'animation d'une machine d'activité nécessite la création d'un ensemble d'utilisateurs qui peuvent exécuter ses actions et l'affectation de ces utilisateurs à des rôles. La plateforme B4MSecure permet de modéliser les affectations définies dans le diagramme de la figure 4.15 et de faire leur traduction en B. Ces affectations peuvent être ainsi effectuées à l'aide de la machine *UserAssignments* de la figure 4.7 (en

utilisant ses opérations). Trois utilisateurs ont été créés : *Bob*, *Paul* et *Jack*. *Bob* est affecté aux deux rôles *Initiator* et *Participant*. L'utilisateur *Paul* est affecté au rôle *Participant*. *Jack* est affecté au rôle *Initiator*.

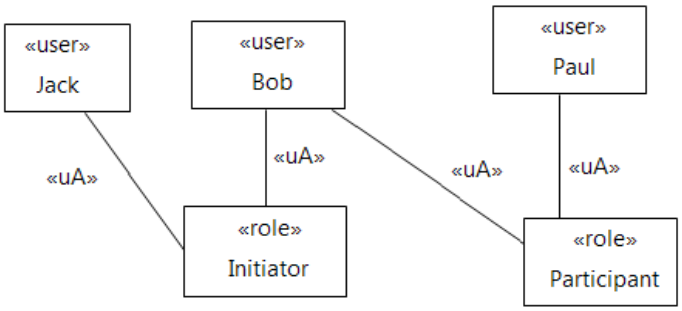

Figure 4.15. Affectation des utilisateurs aux rôles

Une action concrète d'une activité BAAC@UML appelle une opération d'une instance de la classe. La définition des objets appelés par les actions concrètes est donc nécessaire pour l'animation des machines d'activité. La figure 4.16 présente un ensemble d'instances de classes avec leurs liens, qui seront utilisées dans l'animation de la machine d'activité *FollowAnswer*. Nous avons défini deux instances pour chacune des classes *Meeting* (M1 et M2), *Invitation* (I1 et I2) et *Change* (C1 et C2), et trois instances de la classe Person (Bob, Paul et Jack). Nous avons également spécifié les liens entre ces instances. Les instances de classes *Meeting*, *Invitation*, *Person* et *Change* et leurs liens sont définis en initialisant les variables correspondantes dans la machine *Functional* (figure 4.6). La figure 4.17 présente l'initialisation correspondant à la figure 4.16.

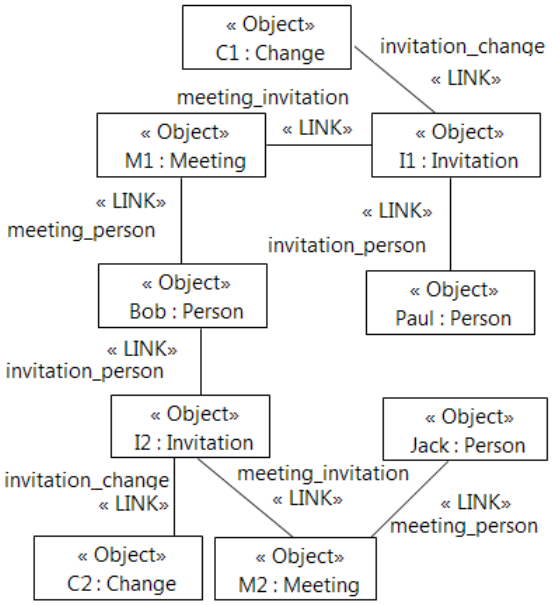

Figure 4.16. Les instances de classes

```
Change := {C1,C2}
||  Person := {Bob,Paul,Jack}
||  Invitation := {I1,I2}
||  Meeting := {M1,M2}
||  meeting_invitation := {(I1↦M1),(I2↦M2)}
||  meeting_person := {(M1↦Bob),(M2↦ Jack)}
||  invitation _change := {(C1↦I1),(C2↦I2)}
||  invitation_person := {(I1↦Paul),(I2↦Bob)}
```

Figure 4.17. Initialisation des classes et leurs liens dans la machine Functional

4.4.2.2 Animation de la machine d'activité

L'animation de la machine d'activité permet de tester l'exécution des scénarios nominaux de l'activité par un utilisateur connecté à un ou plusieurs rôles. Dans l'animation, nous considérons la machine d'activité TBAM (figures 4.13 et 4.14) et les scénarios nominaux *S1*, *S2*, *S3* et *S4* présentés précédemment (2ème paragraphe de la section 4.4.2).

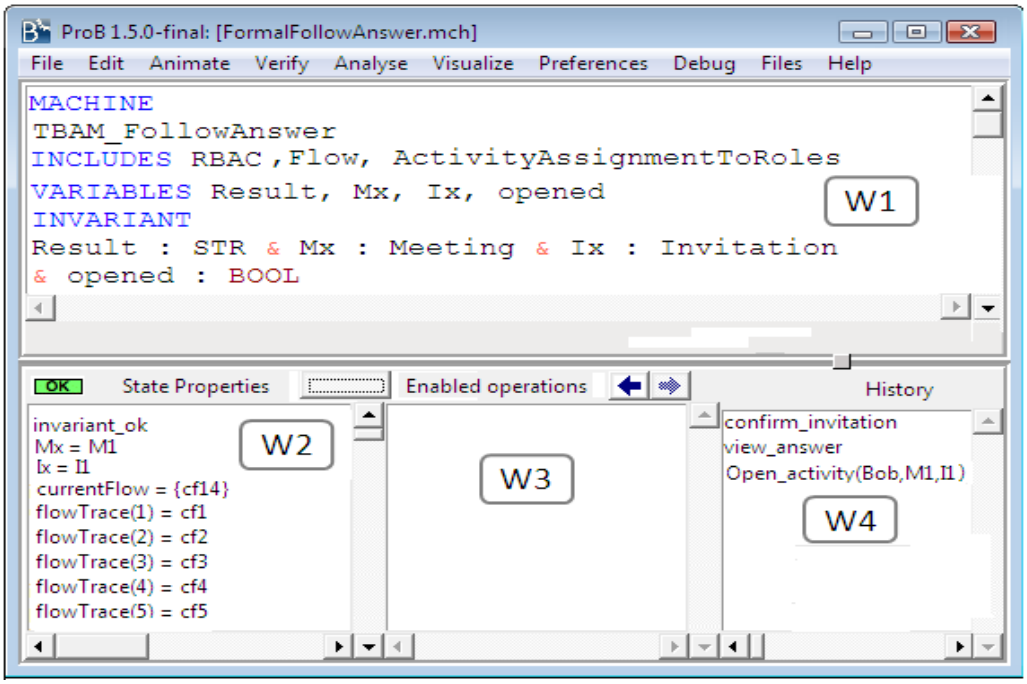

Figure 4.18. Animation de la machine d'activité TBAM avec ProB

La figure 4.18 donne une capture d'écran de l'animation de la machine d'activité *TBAM_FollowAnswer* avec l'outil *ProB*. La fenêtre principale (W1 dans la figure 4.18) affiche la machine *TBAM_FollowAnswer* à animer. La fenêtre *History* (W4) présente la trace d'animation courante. La fenêtre *State Properties* (W2) donne la

valeur actuelle de chaque variable de la machine *TBAM_FollowAnswer*. Enfin, la fenêtre *Enabled Operations* (W3) liste les opérations qui peuvent être appelées à ce stade et dont l'exécution satisfera leur précondition, et préservera l'état invariant.

La fenêtre *History* (W4) de la figure 4.18 présente un exemple de test positif. L'utilisateur *Bob* connecté au rôle *Initiator* exécute avec succès le scénario nominal *S1* de l'activité *Follow answer* avec les instances *M1* de la classe *Meeting* et *I1* de la classe *Invitation*. La fenêtre *State Properties* (W2) montre la trace du flux de contrôle.

4.4.2.3 Encapsulation des scénarios

Afin d'organiser l'animation des machines d'activité et éviter d'animer manuellement chaque étape du même scénario à plusieurs reprises, nous avons défini la machine *Test* de la figure 4.19 à partir de la machine *TBAM_FollowAnswer* de la figure 4.13.

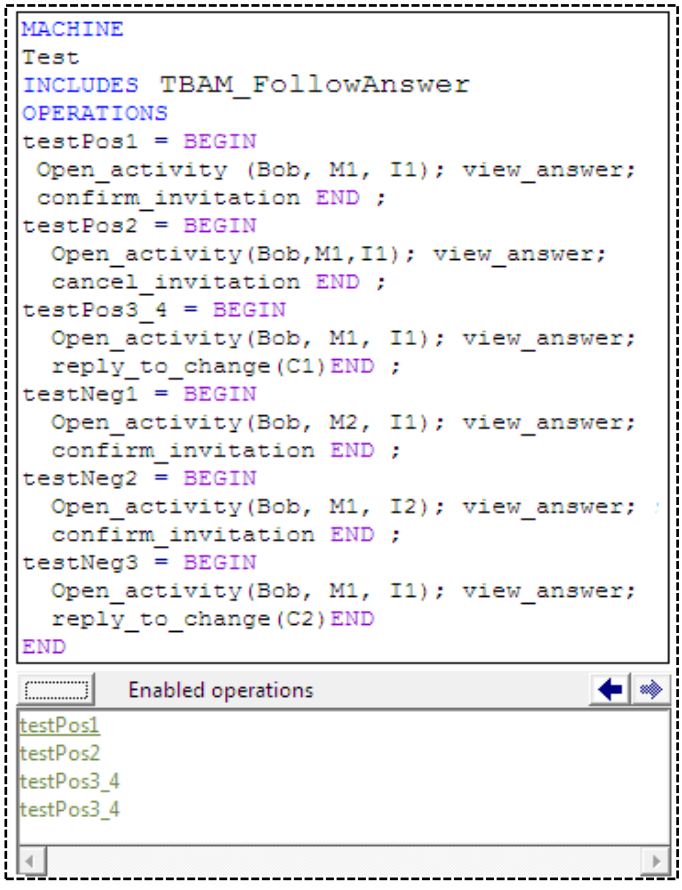

Figure 4.19. La machine Test

La machine *Test* est définie par un ensemble d'opérations où chacune fait appel à une séquence d'opérations (tâches de la machine d'activité) qui représente un scénario nominal d'activité. L'objectif de cette encapsulation est de déterminer les scénarios autorisés et les scénarios interdits par la politique RBAC dans l'animation de la machine d'activité. Un scenario qui fait appel à une opération interdite par la politique de sécurité ne sera pas visible dans l'animation de la machine *Test.*

Dans la figure 4.19, les opérations *testPos1*, *testPos2* et *testPos3_4* représentent des exemples de tests positifs, alors que les opérations *testNeg1*, *testNeg2* et *testNeg3* spécifient des exemples de tests négatifs. Ces dernières ne sont pas activées dans la fenêtre *Enabled Operations*.

4.4.2.4 Tests positifs

Ces tests vérifient la réaction de la politique de sécurité vis-à-vis d'un comportement normal du système exprimé par des scénarios nominaux valides. La politique de sécurité exprimée au niveau de la machine d'activité BAAC@UML ne doit pas bloquer l'exécution des scénarios nominaux par un utilisateur légitime. L'exécution de chacun de ces *scénarios* à l'aide de l'outil *ProB* doit se terminer par un succès.

Dans la fenêtre *Enabled operations* de la figure 4.19, l'animateur permet à l'utilisateur *Bob* d'exécuter les opérations suivantes avec les instances *M1*, *I1* et *C1*:

(i) *testPos1* qui représente le scénario *S1* de l'activité *Follow answer*,
(ii) *testPos2* qui représente le scénario *S2*
(iii) *testPos3_4* qui représente les scénarios *S3* et *S4*.

testPos3_4 apparaît deux fois dans la fenêtre *Enabled operations* pour tenir compte du possible choix de l'initiateur.

Cela signifie que le modèle BAAC@UML de la figure 4.12 exprimé par la machine d'activité de la figure 4.13 n'empêche pas un utilisateur légitime d'exécuter les scénarios nominaux et que les préconditions de contrôle d'accès peuvent être évaluées à *vrai*. L'utilisateur *Bob* est connecté au rôle de l'activité (Initiator) et il respecte la précondition globale de contrôle d'accès et les deux préconditions locales *FA_ChangeAC* et *FA_InvitationAC*.

Les tests positifs signifient également que le filtre formel RBAC, généré à partir de modèles SecureUML et exprimé au niveau des opérations sécurisées appelées par les actions concrètes des tâches, ne bloque pas un utilisateur légitime.

4.4.2.5 Tests négatifs

Ces tests vérifient les interdictions de la politique de sécurité durant l'exécution des scénarios nominaux d'une activité. Ils consistent à spécifier des cas de violation de préconditions de contrôle d'accès pour vérifier que ces préconditions peuvent être évaluées à *faux*. Nous changeons les utilisateurs et leurs rôles ou les objets invoqués par les actions concrètes de l'activité pour spécifier les cas de violation. Après le changement, les scénarios doivent être bloqués à cause des conditions d'accès et non des résultats fonctionnels. En effet, ce type de test permet de vérifier les comportements interdits de la politique de sécurité en invalidant l'un de ses éléments.

Par exemple:

A. L'opération *testNeg1* de la machine *Test* de la figure 4.19 est un exemple de test négatif de la précondition globale de contrôle d'accès (GACPreCondition). L'utilisateur *Bob* connecté au rôle *Initiator* exécute le scénario nominal *S1* de l'activité BAAC@UML de la figure 4.12 avec les instances *M2* de la classe *Meeting* et *I1* de la classe *Invitation*.

   Dans la fenêtre *Enabled operations* de la figure 4.18, l'animateur ne permet pas d'exécuter l'opération *testNeg1*. Le scénario *S1* est bloqué en raison de la violation de l'invariant *meeting_person(Mx)=currentUser* qui formalise GACPreCondition (*Bob* n'est pas le créateur de *M2* (voir figure 4.16). Cela signifie que GACPreCondition peut être évaluée à *faux* ce qui permet de bloquer tout utilisateur qui ne respecte pas la contrainte contextuelle exprimée par cette précondition.

B. L'opération *testNeg2* de la machine *Test* de la figure 4.19 est un exemple de test négatif de la précondition locale de contrôle d'accès *FA_InvitationAC*. L'utilisateur *Bob* connecté au rôle *Initiator* exécute le scénario *S1* de l'activité BAAC@UML avec les instances *M1* de la classe *Meeting* et *I2* de la classe *Invitation*.

   Dans la figure 4.19, l'animateur ne permet pas d'exécuter l'opération *testNeg2*. Dans ce cas, le scénario *S1* est bloqué parce que la précondition locale *FA_InvitationAC* n'est pas vérifiée. L'utilisateur *Bob* vient de lire les informations de l'invitation *I2* qui n'est pas associée avec sa réunion (M1) (voir figure 4.16). Cela signifie que la précondition *FA_InvitationAC* peut être évaluée à *faux*.

C. L'opération *testNeg3* de la machine *Test* de la figure 4.19 est un exemple de test négatif de la précondition locale de contrôle d'accès *FA_ChangeAC*. L'utilisateur *Bob* connecté au rôle *Initiator* exécute les scénarios nominaux S3 et S4 de l'activité BAAC@UML avec les instances *M1* de la classe *Meeting*, *I1* de la classe *Invitation* et *C2* de la classe *Change*.

   Dans la fenêtre *Enabled operations* de la figure 4.19, l'animateur ne permet pas d'exécuter l'opération *testNet3* parce que l'utilisateur *Bob* ne respecte pas la précondition *FA_ChangeAC*; il vient de lire les informations du changement *C2* qui n'est pas associé à l'invitation (I1) de sa réunion (M1) (voir la figure 4.16). Cela signifie que la précondition *FA_ChangeAC* peut être évaluée à *faux*.

Les tests positifs et négatifs vérifient également que le modèle BAAC@UML de la figure 4.12 garantit les différentes règles de politique RBAC :

- Une réunion ne peut être lue ou modifiée que par son créateur
- Un initiateur peut seulement lire, supprimer et modifier les invitations de ses réunions.
- Un initiateur peut lire et répondre uniquement aux propositions de changement associées à une invitation de sa réunion seulement.

Le modèle BAAC@UML autorise les utilisateurs légitimes à jouer les scénarios d'activité et empêche tout autre utilisateur non autorisé par la politique RBAC.

4.5 Conclusion

Dans ce chapitre, nous avons présenté une approche de spécification et de validation formelles de nos modèles d'activités BAAC@UML en cherchant les incohérences entre ces modèles et les modèles SecureUML qui expriment la vue statique d'une politique RBAC.

Nous avons commencé par la transformation de nos modèles BAAC@UML qui expriment une politique de contrôle d'accès au niveau des activités d'un BP en machines B à l'aide de la plateforme B4MSecure. Les machines B considèrent les spécifications formelles issues du diagramme de classes du système et de modèles SecureUML. Nous avons proposé deux types de machines B. Le premier type, appelé ABAM, représente les actions concrètes d'une activité BAAC@UML par des opérations B. Il permet de tester la politique de contrôle d'accès au niveau de chaque action concrète de l'activité. Cependant il est difficile de tester ce type de machines pour des activités de grande taille. Le deuxième type, appelé TBAM, représente les

tâches d'une activité BAAC@UML par des opérations B. Il permet de tester la politique de contrôle d'accès dans l'exécution de différentes tâches de l'activité.

Après la dérivation, nous avons tiré profit de l'animateur *ProB* et du prouveur *Atelier B* pour vérifier la cohérence des modèles et leurs propriétés fonctionnelles et de sécurité.

Dans l'animation, nous avons réalisé une cinquantaine de tests, entre positifs et négatifs, pour chacun des six modèles BAAC@UML décrivant les six cas d'utilisation de la figure 2.2 (Chapitre 2). Nous avons systématiquement construit ces tests pour chaque modèle BAAC@UML. Ces tests nous ont permis d'identifier et de corriger les erreurs dans les modèles BAAC@UML et les incohérences entre ces modèles et les modèles SecureUML.

La preuve, en utilisant *Atelier B,* nous a permis de prouver la conformité des propriétés d'invariance des machines d'activités en assurant que les contraintes *assignedRoles* et *GACPreCondition* qui ont été exprimées comme invariants sont conservées dans l'exécution de l'activité.

Chapitre 5

Mise en œuvre des modèles BAAC@UML dans des outils

5.1 Introduction

Notre démarche de spécification et de V&V de politiques RBAC au niveau des activités d'un BP utilise les standards de l'IDM tels que UML. Nous souhaitons également que nos outils s'appuient sur les standards de cette approche. La plateforme logicielle *EMF*[12] (Eclipse Modeling Framework) est le standard de facto qui sert de support à l'IDM. Cette plateforme de la fondation *Eclipse*[13] fournit un ensemble d'outils pour créer des modèles et leurs méta-modèles, et faire leur vérification et leur transformation. Elle supporte les différents standards de développement logiciel tels que le langage *Java*[14], *XML* (XML - Extensible Markup Language) et *UML*. Elle permet également de réaliser des transformations entre ces notations. Le méta-modèle *Ecore* est utilisé pour représenter les modèles EMF.

Dans le cadre de notre étude, nous avons développé trois outils basés sur la plateforme EMF : *Graphical-ACP*, *ACP-Consistency et Extended-B4MSecure*. *Graphical-ACP* permet la représentation graphique des modèles UML de contrôle d'accès (BAAC@UML et SecureUML). *ACP-Consistency* sert à vérifier la cohérence entre les modèles BAAC@UML et SecureUML. *Extended-B4MSecure* étend la plateforme B4MSecure pour la génération de spécifications formelles, exprimées dans le langage B, à partir des modèles BAAC@UML.

La section 5.2 discute la mise en œuvre de l'outil *Graphical-ACP*. La section 5.3 montre les étapes de développement de l'outil *ACP-Consistency*. La section 5.4 décrit les extensions de la plateforme B4MSecure pour la traduction des modèles BAAC@UML en B. Enfin, la dernière section présente la démarche d'utilisation de nos outils.

5.2 Graphical-ACP

Dans ce travail, nous avons utilisé l'environnement *Papyrus*[15] , disponible sous la forme d'un plugin de la plateforme EMF, pour la mise en œuvre de l'outil *Graphical-ACP*. Papyrus est un environnement logiciel open-source destiné à l'IDM. Il supporte le langage UML et d'autres langages de modélisation connexes tels que RobotML, UML-RT, SysML et MARTE, et il fournit plusieurs techniques pour la modélisation, la simulation de modèles, la vérification de modèles, l'analyse de la sécurité et de la performance, la génération de code ainsi que d'autres techniques.

[12] *https://eclipse.org/modeling/emf/*
[13] *https://eclipse.org/*
[14] *https://www.java.com/*
[15] *https://eclipse.org/papyrus/*

Papyrus offre également un support très avancé pour les profils UML qui permet aux utilisateurs de définir des éditeurs pour les DSL (Domain Specific Language) basés sur le standard UML2 (Ouhammou, Keulkeul, 2012). De ce fait, chaque partie de l'environnement Papyrus peut être personnalisée: l'explorateur de modèle, la notation et le style des diagrammes, la palette de menus, etc. Nous avons tiré profit de ces capacités de l'environnement *Papyrus* pour la mise en œuvre de l'outil *Graphical-ACP* qui permet la représentation graphique des modèles BAAC@UML et SecureUML.

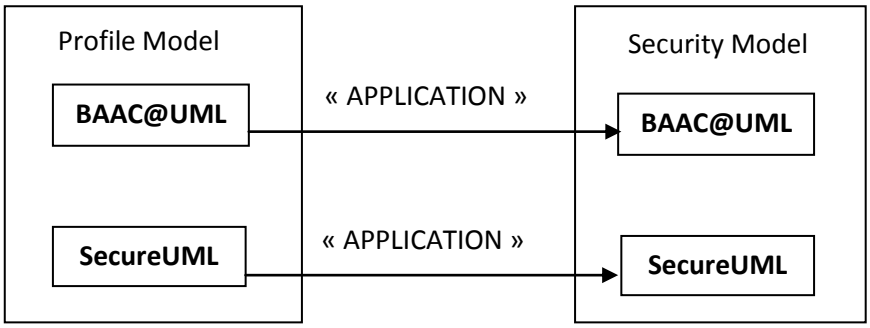

Figure 5.1. Mise en œuvre de l'outil Graphical-ACP

La mise en œuvre de notre outil *Graphical-ACP* a été réalisée en deux étapes (voir la figure 5.1) :

- *Dans la première étape*, nous avons créé le modèle de profil qui spécifie le méta-modèle BAAC@UML, présenté dans la figure 2.10, et le méta-modèle SecureUML de la figure 2.1. Le modèle de profil définit les stéréotypes représentant les concepts de deux profils (comme par exemple ConcreteActivity du profil BAAC@UML et Permission du profil SecureUML) et lie ces stéréotypes aux méta-classes UML correspondantes (comme par exemple ConcreteActivity est lié à la méta-classe Activity, Permission est lié à la méta-classe AssociationClass) en utilisant *la relation extension*. Il définit également les liens de chaque stéréotype avec les autres stéréotypes ou méta-classes UML. Avant UML2, les stéréotypes pouvaient être librement créés sur les éléments des diagrammes UML. Depuis UML2, les stéréotypes ne peuvent être définis que dans un profil.
- *Dans la deuxième étape*, nous avons appliqué le modèle de profil sur les diagrammes UML pour permettre l'utilisation des stéréotypes définis. Cela nous a permis de créer un modèle qui spécifie les différents diagrammes BAAC@UML (comme ceux des figures 3.12 (Chapitre 3) et 2.6 (Chapitre 2)) et SecureUML (comme ceux des figures 2.4 (Chapitre 2) et 3.11 (Chapitre 3)) de notre étude de cas du système d'organisation de réunions.

5.3 ACP-Consistency

Nous nous sommes basés sur l'environnement *EcoreTools*[16] de la plateforme EMF pour la mise œuvre de l'outil *ACP-Consistency* qui vérifie la cohérence entre les modèles BAAC@UML de spécification de la vue dynamique de politiques RBAC et les modèles SecureUML de spécification de la vue statique de politiques RBAC. *EcoreTools* permet de définir un méta-modèle associé à des contraintes OCL, d'en créer des instances et de vérifier les contraintes OCL sur ces instances.

La figure 5.2 explique la mise en œuvre de l'outil *ACP-Consistency*. Nous avons commencé par la spécification des différentes opérations et invariants OCL, présentés dans le chapitre 3, à l'aide du langage *OCLinEcore* intégré dans l'environnement EcoreTools. Les contraintes OCL ont été associées aux différentes méta-classes des méta-modèles BAAC@UML et SecureUML. OCLinEcore est un environnement d'édition riche qui rend à la fois Ecore et OCL accessibles aux utilisateurs. Il supporte l'utilisation du langage OCL intégré dans des annotations pour tous les concepts UML liés à la classe. Il offre une vue textuelle cohérente avec un méta-modèle Ecore et il assure une vérification sémantique des expressions OCL.

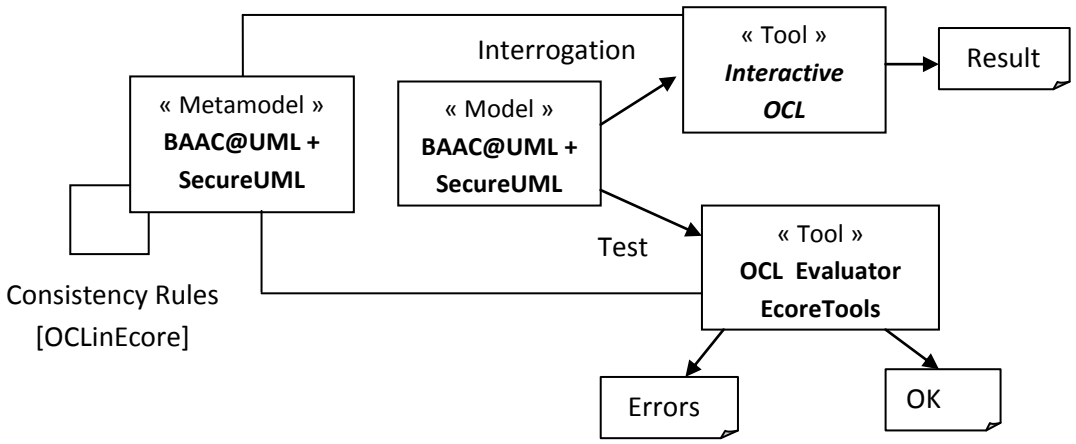

Figure 5.2. Mise en œuvre de l'outil ACP-Consistency

EcoreTools permet ensuite de créer des instances correspondant aux modèles SecureUML et aux modèles BAAC@UML. Après la création des instances, l'évaluateur OCL de *EcoreTools* permet de détecter les incohérences entre les modèles BAAC@UML et SecureUML et d'afficher les invariants violés.

L'outil *Interactive OCL* permet d'interroger les instances d'un méta-modèle et d'évaluer des expressions exprimées en OCL. Nous avons utilisé cet outil pour tester

[16] *http://www.eclipse.org/ecoretools/*

et valider les opérations et les invariants OCL présentés dans le chapitre 3 avant leur association aux méta-classes de notre méta-modèle.

5.4 Extended-B4MSecure

B4MSecure[17] est une plateforme logicielle d'Eclipse qui permet la traduction automatique en spécifications B des diagrammes de classes fonctionnels et des modèles SecureUML. Dans le cadre de notre étude, nous avons étendu cette plateforme pour la traduction des modèles BAAC@UML en spécifications B. Cette extension a été réalisée en trois étapes. Les sections 5.4.1, 5.4.2 et 5.4.3 expliquent ces étapes. Dans le cadre de cette extension, nous avons considéré les modèles BAAC@UML d'activité concrète (voir la section 2.3.3 du chapitre 2) et l'approche ABAM (voir la section 4.3.3.3.1 du chapitre 4).

5.4.1 Extension du méta-modèle B4MSecure

La plateforme B4MSecure est basée sur un méta-modèle. Nous avons commencé notre extension de la plateforme par l'intégration des concepts de notre profil BAAC@UML dans ce méta-modèle. Le diagramme de la figure 5.3 présente les éléments du méta-modèle mis en place dans la plateforme B4MSecure en lien avec les concepts du profil BAAC@UML.

Le méta-modèle de la plateforme B4MSecure comprend les concepts du modèle RBAC enrichi par la notion d'organisation qui exprime l'appartenance des utilisateurs à des organisations et la notion de délégation de rôles qui offre la possibilité de déléguer des rôles à des utilisateurs. Dans ce méta-modèle une permission autorise un rôle à accéder aux éléments d'une classe. Cette autorisation est associée éventuellement à une contrainte. Une permission contient des actions de type *MethodAction* qui permettent l'accès à des opérations de la classe cible de la permission. Elle contient également des actions abstraites, de type *EntityAction*, qui donnent le droit d'accès aux éléments de la classe cible de la permission, comme par exemple les terminaisons d'associations et les attributs. Le méta-modèle définit l'association *superRoles* pour spécifier la hiérarchie des rôles en permettant aux rôles supérieurs d'hériter des permissions des sous-rôles. La méta-classe *Session* représente la connexion des utilisateurs au système. Au moyen d'une session, un utilisateur active un ensemble de rôles parmi ceux qui lui sont affectés. L'appartenance des utilisateurs à des organisations et la possibilité de déléguer des rôles à des utilisateurs sont représentés respectivement par les méta-classes *Organization* et *DelegateRole*.

[17] *http://b4msecure.forge.imag.fr/*

Les concepts du modèle BAAC@UML d'activité concrète sont : *ConcreteActivity*, *ConcreteAction* et *LACPreCondition*. Une activité concrète est affectée à un ou plusieurs rôles. Elle est composée d'un ensemble d'actions concrètes où chacune fait appel à une opération de classe. Certaines de ces actions sont associées à une précondition locale de contrôle d'accès qui fait référence à une ou plusieurs contraintes d'autorisation.

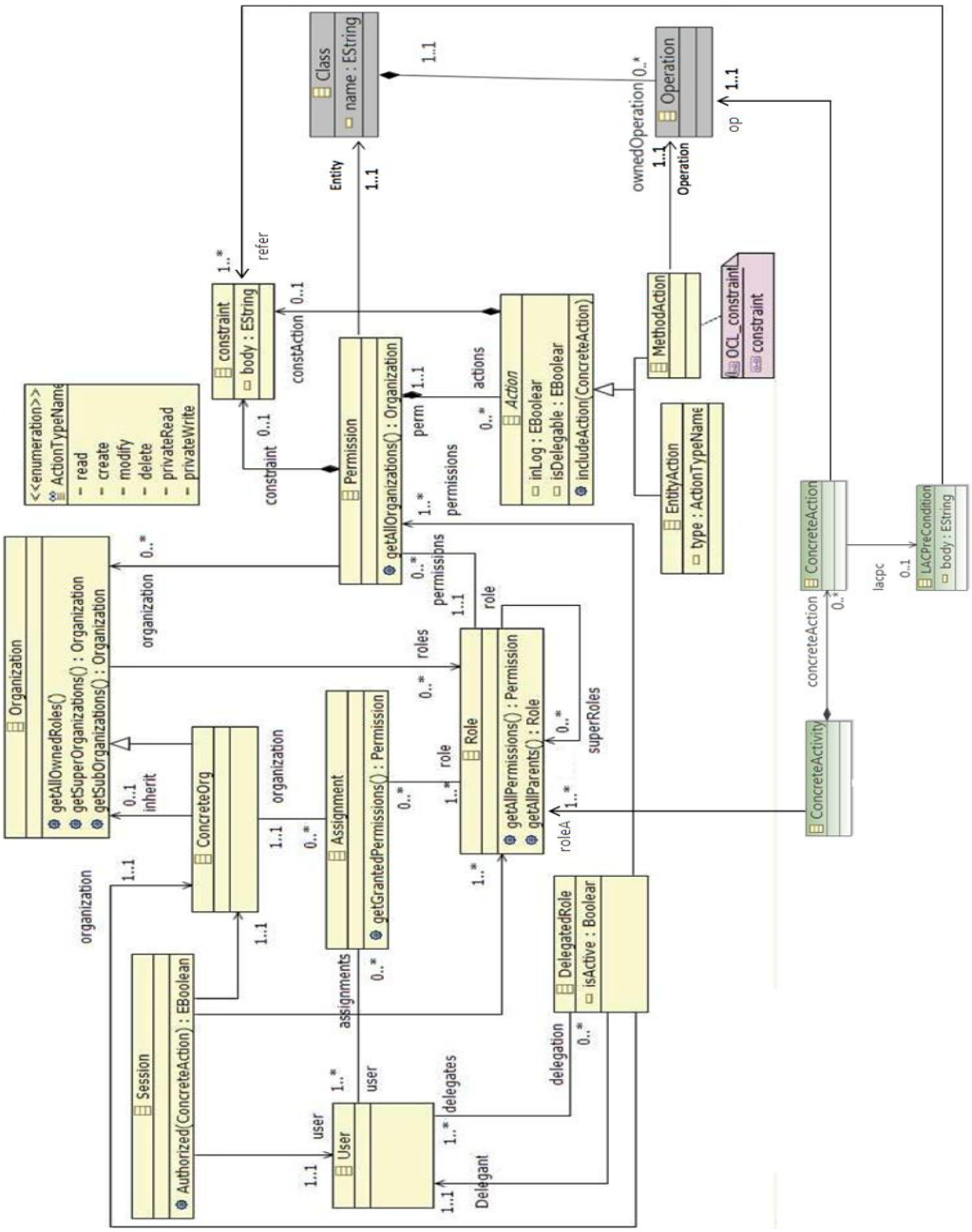

Figure 5.3 : Méta-modèle étendu de B4MSecure

5.4.2 Des modèles BAAC@UML aux instances du méta-modèle B4MSecure

La syntaxe concrète des modèles BAAC@UML est basée sur des éléments d'un diagramme d'activités stéréotypé. Notre outil *Graphical-ACP* permet la représentation graphique de ces modèles qui correspondent à une instance du méta-modèle UML enrichi par un profil. Pour qu'on puisse dériver les modèles BAAC@UML en B à l'aide de la plateforme B4MSecure, il faut que nos modèles BAAC@UML soient conformes au méta-modèle étendu de la plateforme B4MSecure présenté dans la section précédente. Pour ce faire, nous avons implémenté des règles de transformation Modèle-vers-Modèle (M2M) qui permettent d'obtenir une instance de méta-modèle étendu de la plateforme B4MSecure à partir d'une instance de méta-modèle UML stéréotypé (voir la figure 5.4).

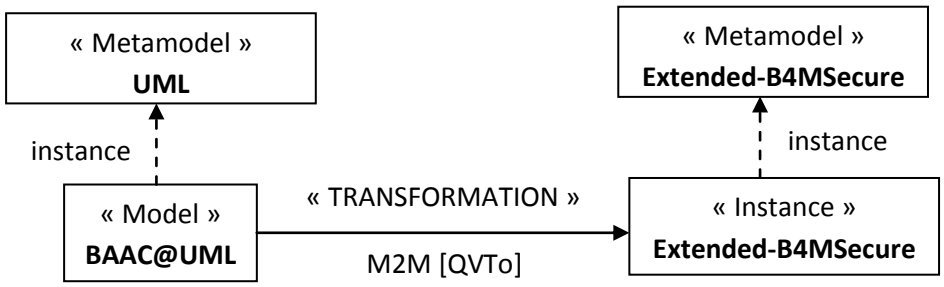

Figure 5.4. Transformation M2M des modèles BAAC@UML

QVT (Query/View/Transformation) est un langage standardisé, défini par l'OMG en 2007, pour exprimer des transformations de modèles. Il permet de transformer des modèles source en modèles cible. QVT définit trois langages nommés respectivement *QVT-Relation*, *QVT-Core* et *QVT-Operational*. Ces langages sont organisés en une architecture en couches. *QVT-Relation* et *QVT-Core* sont des langages déclaratifs à deux niveaux différents d'abstraction. *QVT-Operational* est un langage hybride qui propose une structure déclarative et permet l'utilisation d'expressions impératives (comme par exemple les boucles et les conditions).

Dans le cadre de notre travail, nous nous sommes basés sur le langage *QVT-Operational* pour implémenter les règles qui transforment nos diagrammes d'activités en instances du méta-modèle étendu de B4MSecure. Pour réaliser la transformation, nous avons opté pour l'outil *SmartQvt*. Cet outil compile les règles de transformation spécifiées dans QVTo pour produire du code Java utilisé pour la mise en œuvre de cette transformation. Il est disponible sous forme de plugin de la plateforme EMF et se compose de trois éléments:

- *QVT Editor:* il permet d'écrire les spécifications QVT.

- *QVT Parser:* il permet de convertir la syntaxe concrète textuelle en une représentation correspondant au méta-modèle QVT.
- *QVT Compiler:* il permet la mise en œuvre des transformations.

5.4.3 Dérivation des modèles BAAC@UML en B

Après la mise en œuvre des règles de transformation qui permet la génération automatique d'une instance de méta-modèle étendu de B4MSecure à partir des modèles BAAC@UML, cette étape se focalise sur la génération des spécifications B à partir de cette instance. Dans les spécifications B des modèles BAAC@UML, les actions concrètes font appel aux opérations sécurisées de la machine *RBAC* qui encapsulent des opérations de la machine *Functional*. Il faut donc analyser la trace de la transformation des diagrammes de classes fonctionnels et des modèles SecureUML et retrouver les éléments de leur traduction. En effet, cette étape correspond à une transformation Modèle-vers-Texte (M2T) qui prend en entrée les instances de méta-modèle étendu de B4MSecure et les traces générées lors de la traduction de modèles SecureUML et du diagramme de classes, et produit en sortie un code textuel exprimé dans le langage B (voir la figure 5.5).

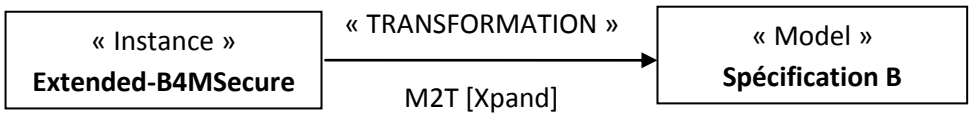

Figure 5.5. Transformation M2T des modèles BAAC@UML

Pour réaliser la transformation M2T, nous avons utilisé la template *Xpand* qui fournit un langage déclaratif, mélange de java et OCL. Cette template a été proposée dans le cadre du projet *openArchitectureWare* (oAW) et elle est disponible actuellement sous forme de plugin de la plateforme EMF. *Xpand* est utile dans des contextes différents de l'IDM, comme par exemple, la validation, l'extension des méta-modèles, la génération de code, la transformation du modèle, etc. Elle fournit une couche uniforme d'abstraction qui supporte plusieurs méta-méta-modèles, comme par exemple, EMF, Ecore, UML2, etc. La template *Xpand* inclut également un éditeur qui fournit des fonctionnalités telles que le débogage, la navigation, la complétion de code, etc.

5.5 Conclusion

Dans cette conclusion, nous présentons une démarche pour les utilisateurs de nos outils. Nous commençons par la modélisation graphique en UML du diagramme de classes fonctionnel, et des vues statique (avec SecureUML) et dynamique (avec

BAAC@UML) d'une politique RBAC à l'aide de notre outil *Graphical-ACP*. Nous proposons la structuration indiquée dans la figure suivante.

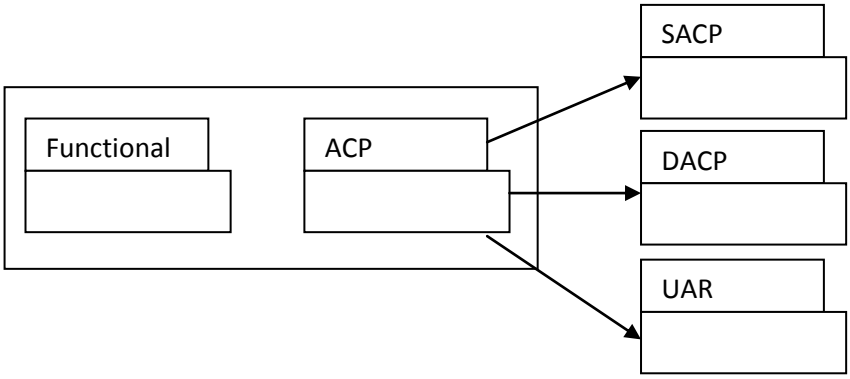

Figure 5.6. Structuration des modèles

Le package *Functional* pour représenter le diagramme de classes fonctionnel. Le package *ACP* pour représenter les modèles UML de spécification de politiques de contrôle d'accès. Ce package inclut trois packages : SACP qui spécifie les modèles SecureUML, DACP qui spécifie les modèles BAAC@UML et UAR qui spécifie l'affectation des utilisateurs aux rôles.

Après la représentation graphique de la politique RBAC, nous tirons profit de l'outil *ACP-Consistency* pour vérifier la cohérence structurelle entre les modèles BAAC@UML encapsulés dans le package DACP et les modèles SecureUML encapsulés dans le package SACP.

Après le traitement des incohérences, nous utilisons la version que nous avons étendue de la plateforme B4MSecure (Extended-B4MSecure) pour la traduction des modèles graphique de contrôle d'accès en B. Nous commençons par la traduction du modèle fonctionnel puis nous générons les instances du méta-modèle étendu de la plateforme B4MSecure. Après la génération de ces instances, nous traduisons les modèles SecureUML puis les modèles BAAC@UML en B.

Enfin, la dernière étape de notre approche est consacrée à la V&V des machines B issues de modèles UML de spécification de politiques RBAC.

L'ensemble des outils développés dans le cadre de notre étude sont disponibles en open source. Dans le futur nous prévoyons de publier ces outils et de les rendre accessibles chez les industriels et les académiques.

Conclusion générale

C.1 Bilan de notre contribution

Dans cette étude, nous avons proposé une approche qui combine les langages UML et B pour la spécification et la validation de politiques RBAC au niveau des activités d'un BP. Nous avons commencé par la spécification semi-formelle des règles de contrôle d'accès à l'aide de notre profil BAAC@UML. Les modèles d'activités sont ensuite traduits en une spécification formelle exprimée dans le langage B. Cette formalisation nous a permis de définir un cadre rigoureux favorisant la V&V de la politique RBAC. Notre approche fournit un certain nombre d'avantages comprenant:

- La séparation entre la spécification des BPs et la spécification de la politique de contrôle d'accès. Notre approche commence par la spécification des activités métiers fonctionnelles. Ensuite, des gardes de contrôle d'accès sont associées à ces activités.
- La spécification graphique et formelle d'une politique RBAC au niveau des activités d'un BP. Les modèles graphiques facilitent la compréhension de la politique RBAC. La spécification formelle des modèles permet la validation de la politique RBAC.
- Les modèles BAAC@UML prennent en compte le modèle fonctionnel et la spécification statique de la politique RBAC exprimée par les modèles SecureUML. Cela permet d'éviter les contradictions dans le la mise en œuvre de la politique RBAC exprimée par les modèles BAAC@UML.
- La preuve et l'animation de la spécification formelle des modèles BAAC@UML. Cela vérifie la cohérence des modèles et que ces modèles répondent aux besoins de l'analyste.

Dans le cadre de la spécification semi-formelle, nous avons proposé le profil BAAC@UML qui adapte les diagrammes d'activités d'UML2 afin de contrôler l'accès des utilisateurs aux activités d'un BP. Le modèle BAAC@UML spécifie le comportement de différentes tâches de l'activité par une coordination d'actions concrètes qui font référence à des opérations sur les objets des classes. Ensuite, une politique de contrôle d'accès est exprimée au niveau des activités pour contrôler l'accès des utilisateurs à leurs actions. Dans cette politique, nous considérons l'affectation des utilisateurs aux rôles, la hiérarchie des rôles, et les contraintes contextuelles d'autorisation (exprimées par les préconditions de contrôle d'accès) qui font référence au modèle fonctionnel ainsi qu'à l'utilisateur et ses rôles. La démarche que nous proposons construit graduellement une politique de contrôle d'accès et

identifie les gardes qui la mettront en œuvre dynamiquement dans le déroulement des activités. Cette représentation est proche des scénarios d'exécution et facilite l'implémentation de la politique de sécurité sur des plateformes logicielles. Nous avons, en outre, appliqué notre démarche sur le système d'organisation de réunions, et développé ainsi tous les cas d'utilisation de la figure 2.2. Les règles de contrôle d'accès ont ainsi été exprimées sur les différentes activités du système. Cette étude de cas nous a permis de montrer l'intérêt d'exprimer des règles de contrôle d'accès au niveau des BPs. Des études de cas de plus grande taille seront entreprises dans des futurs travaux.

La séparation entre les préoccupations fonctionnelles et sécuritaires est au cœur de notre approche. Cette séparation des préoccupations vise à maîtriser la complexité du système, et peut également se montrer utile quand il s'agit de réaliser la sécurisation d'un système d'information préexistant. De plus, elle facilite l'évolution de la politique de contrôle d'accès. Dans notre approche, les aspects fonctionnels sont exprimés dans les cas d'utilisation, le diagramme de classes et les diagrammes d'activités abstraits, et leur raffinement dans l'enchaînement des actions concrètes. Les aspects sécuritaires sont exprimés statiquement, sous forme de permissions associées à des rôles, pour chaque classe protégée du diagramme de classes. A cet effet, nous nous basons sur le profil SecureUML. Les aspects sécuritaires sont ensuite exprimés au niveau des modèles BAAC@UML par des préconditions associées localement ou globalement aux activités concrètes. En outre, les contraintes contextuelles d'autorisation permettent des interactions entre les aspects fonctionnels et sécuritaires. Elles sont spécifiées dans la vue statique et prennent la forme de gardes dans la vue dynamique.

Dans ce travail, nous avons proposé un méta-modèle, associé à des contraintes OCL, qui spécifie la sémantique de nos modèles d'activités et la sémantique de leurs relations avec les modèles SecureUML. L'ensemble des règles de cohérence définies dans ce travail permet d'éviter toute contradiction entre la représentation statique de la politique RBAC exprimée en SecureUML et la représentation dynamique exprimée en BAAC@UML. Ces règles sont supportées par un outil. Elles sont illustrées par l'étude de cas du système d'organisation de réunions où elles ont permis de corriger les erreurs de cohérence détectées dans les modèles de contrôle d'accès des vues statique et dynamique.

Dans le cadre de cette étude, nous avons présenté une approche qui permet de traduire les modèles BAAC@UML en une spécification B et d'utiliser cette spécification pour vérifier les contraintes de sécurité des modèles BAAC@UML et leur cohérence sémantique avec les modèles SecureUML. Nous avons étendu la plateforme B4MSecure pour définir les transformations en B. Les fondements IDM

mis en place dans cette plateforme et sa flexibilité nous ont permis de réaliser cette extension. Les machines B issues des modèles BAAC@UML formalisent les éléments fonctionnels (tâche, action concrète et flot de contrôle) et de sécurité (rôle et précondition de contrôle d'accès) de l'activité. Après la dérivation, nous avons tiré profit de l'animateur *ProB* et du prouveur *Atelier B* pour tester les machines d'activités. Le prouveur *Atelier B*, nous a permis de produire des obligations de preuve et de vérifier que les invariants de la machine d'activité sont conservés dans l'exécution de l'activité. L'animateur *ProB* nous a permis de simuler l'exécution des scénarios d'activité par un utilisateur connecté à des rôles. L'animation permet d'évaluer les préconditions contextuelles et de vérifier que la politique RBAC est bien respectée au niveau des modèles d'activités.

Les principaux travaux présentés dans cette thèse ont été publiés dans la conférence *RCIS2016* (http://ieeexplore.ieee.org/document/7549284/?arnumber=7549284) et dans la revue ISI (http://isi.revuesonline.com/article.jsp?articleId=36419).

Notre approche d'IDM a été supportée par des outils qui permettent la représentation graphique des modèles BAAC@UML et SecureUML, la vérification de leur cohérence, et leur traduction en *B*. Ces outils s'appuient sur la plateforme *EMF* et supportent les différents standards de l'approche *MDA*. Ils sont également disponibles en open source.

C.2 Perspectives

Les contraintes *TBSoD* (Task-based Separation of Duty) et *BoD* (Binding of Duty) considèrent l'ordre et l'historique d'une tâche dans une instance particulière de processus pour décider si un certain sujet ou rôle est autorisé à effectuer une certaine tâche (Strembeck, Mendling, 2011) (Botha, Eloff, 2001). L'extension de notre profil pour tenir compte de ces contraintes est une des perspectives importantes de ce travail. Dans le cadre de nos modèles BAAC@UML, certaines tâches ou activités devraient être exécutées par des utilisateurs distincts, par exemple pour éviter un conflit d'intérêts. Les contraintes TBSoD empêchent un sujet d'effectuer deux activités ou deux tâches d'une activité en conflit d'intérêts. Les contraintes BoD définissent un ensemble d'activités ou de tâches qui doivent être exécutées par le même sujet. Avant de s'intéresser à TBSoD et BoD, il faudrait relâcher les contraintes sur les utilisateurs et les rôles dans les diagrammes BAAC@UML. On devrait permettre qu'une activité soit une collaboration de plusieurs utilisateurs, chacun dans des rôles différents.

D'autres extensions peuvent être envisagées en intégrant, par exemple, la délégation des activités et leurs tâches à d'autres utilisateurs, ou en intégrant des

contraintes de contexte (comme par exemple, l'heure, la date ou le lieu d'exécution de l'activité).

Une autre perspective qui ne manque pas d'importance consiste à définir des transformations automatiques, de type *M2M*, entre les modèles BAAC@UML et les modèles SecureUML. Un exemple de transformation est de prendre en entrée les diagrammes fonctionnels d'activités concrètes et les diagrammes SecureUML et de produire en sortie des diagrammes d'activités de contrôle d'accès. Pour ce faire, des règles de transformation doivent être définies et exprimées à l'aide des langages de transformation tels que *QVT* et *ATL*. Après la transformation, on peut imaginer d'avoir des modèles BAAC@UML avec des contraintes sécuritaires (rôles assignés aux activités et préconditions de contrôle d'accès) qui soient plus fortes que la politique de sécurité exprimée par les modèles SecureUML.

Les modèles BAAC@UML présentent une vue proche d'un langage de programmation impératif tels que *C ++* ou *Java*. On peut également définir des transformations de type *M2T* pour la génération de code à partir des modèles BAAC@UML.

Les perspectives liées aux activités de V&V formelles consistent, d'une part, à augmenter l'ensemble des rôles et d'instances de classes et à automatiser le processus V&V en exploitant les capacités de *model-checking* de l'outil *ProB* et, d'autre part, de finaliser la preuve de la cohérence entre nos modèles BAAC@UML et les modèles SecureUML à l'aide du *prouveur interactif* de l'outil *Atelier B.* Dans le cadre des activités de V&V, nous pouvons également tirer profit des travaux de génération des cas de test à partir des diagrammes d'activités d'UML, comme par exemple (Tripathy, Mitra, 2012) et (Mingsong et al.,2006), pour la spécification automatique des scénarios nominaux d'activités et des scénarios de test positifs et négatifs.

Les perspectives liées aux outils consistent à étendre l'outil *Graphical-ACP* pour la représentation graphique des autres modèles UML de spécification de politiques de contrôle d'accès, comme par exemple les modèles *UMLsec*. Une extension de l'outil *ACP-Consistency* peut être également envisagée pour traiter les incohérences entre les nouveaux modèles et les modèles BAAC@UML. Nous pensons également à intégrer les deux outils *Graphical-ACP* et *ACP-Consistency* dans la plateforme B4MSecure.

Bibliographie

- Aalst W. M. P. (2004). *Business Process Management Demystified: A Tutorial on Models, Systems and Standards for Workflow Management*. Lectures on Concurrency and Petri Nets, LNCS, vol. 3098, p. 1-65. Springer Berlin Heidelberg.

- Abou El Kalam A., El Baida R., Balbiani P., Benferhat S., Cuppens F., Deswarte Y., Miège A., Saurel C., Trouessin G. (2003). *Organization Based Access Control*. In Proceedings of IEEE 4th International Workshop on Policies for Distributed Systems and Networks, Lake Como, Italy.

- Abrial J.-R. (1996). *The B-book: Assigning Programs to Meanings*. Cambridge University Press.

- Achouri A., Ben Ayed L. (2014). *UML Activity Diagram to Event-B: A Model Transformation Approach Based on the Institution Theory*. In IEEE 15th International Conference on Information Reuse and Integration, p. 823-829. San Francisco, CA, USA.

- Ahn G., Sandhu R., Kang M., Park J. (2000). *Injecting RBAC to Secure a Web-Based Workflow System*. In the 5th ACM Workshop on Role-Based Access Control, p. 1-10. New York, NY, USA, Morgan Kaufmann Publisher.

- Alghathbar K. (2012). *Representing Access Control Policies in Use Cases*. International Arab Journal of Information Technology, vol. 9, no 3.

- Allaki D., Dahchour M., En-nouaary A. (2015). *A New Taxonomy of Inconsistencies in UML Models with their Detection Methods for Better MDE*. International Journal of Computer Science and Applications, vol. 12, no 1, p. 48 – 65.

- Amálio N., Stepney S., Polack F. (2004*). Formal Proof from UML Models.* Formal Methods and Software Engineering, LNCS, vol. 3308, p. 418–433. Springer Berlin Heidelberg.

- ANSI. (2004). *Role Based Access Control*. American National Standard for Information Technology, vol. 359, no 2004, p. 1-47.

- Armando A., Ponta S. (2010). *Model Checking of Security-Sensitive Business Processes*. Formal Aspects in Security and Trust, LNCS, vol. 5983, p. 66-80. Springer Berlin Heidelberg.

- Arsac W., Compagna L., Pellegrino G., Ponta S. (2011). *Security Validation of Business Processes via Model-Checking*. Engineering Secure Software and Systems, LNCS, vol. 6542, p. 29-42. Springer Berlin Heidelberg.

- Atluri V., Warner J. (2005). *Supporting Conditional Delegation in Secure Workflow Management Systems*. In Proceedings of the Tenth ACM Symposium on Access Control Models and Technologies, p. 49–58. New York, NY, USA.

- Audibert L. (2008). UML 2. Institut Universitaire de Technologie de Villetaneuse – Département Informatique. (http ://laurentaudibert.developpez.com/coursuml/ html/coursuml. html)

- Ayoub M. F., Hassan R., Elmongui H. G. (2012). *ESAC-BPM: Early Security Access Control in Business Process Management*. In Proceedings of the Seventh International Conference on Software Engineering Advances, p. 650–655. Lisbon, Portugal.

- Basin D. A., Clavel M., Doser J., Egea M. (2009). *Automated Analysis of Security-Design Models*. Information and Software Technology, vol. 51, no 5, p. 815-831.

- Basin D. A., Doser J., Lodderstedt T. (2006). *Model driven security: From UML models to access control infrastructures*. ACM Transactions on Software Engineering and Methodology, vol. 15, no 1, p. 39-91.

- Bazex P., Bodeveix J., Millan T., Camus C. L., Percebois C. (2003). *Vérification de modèles UML fondée sur OCL*. In Congrès Inforsid, p. 185–202. Nancy, France.

- Behnia S. (2000). *Test de modèles formels en B : cadre théorique et critères de couvertures*. Thèse de doctorat. Institut National Polytechnique de Toulouse, France.

- Belhaj H. (2012). *Méthode B pour la spécification et la vérification formelle des systèmes répartis ouverts*. Thèse de doctorat. Université Mohammed V – Agdal, Morocco.

- Bell D. E., LaPadula L.(1973). *Secure Computer Systems: A Mathematical Model*. Technical Report, MTR 2547, vol 2, Mitre Corporation, Bedford, Mass.

- Bertino E., Ferrari E., Atluri V. (1999). *The specification and enforcement of authorization constraints in workflow management systems*. ACM Transactions on Information and System Security, vol. 2, no 1, p. 65-104.

- Bolusset T. (2004). *B-Space : raffinement de descriptions architecturales en machines abstraites de la méthode formelle B.* Thèse de doctorat. Université de Savoie, France.

- Borges R. M., Mota A. C. (2007). *Integrating UML and Formal Methods*. Electronic Notes in Theoretical Computer Science, vol. 184, p. 97 – 112.

- Botha R., Eloff J. (2001). *Separation of duties for access control enforcement in workflow environments.* IBM Systems Journal, vol. 40, no 3, p. 666–682.

- Boukhebouze M. (2010). *Gestion du changement et vérification formelle de processus métier : une approche orientée règle*. Thèse de doctorat. INSA de Lyon, France.

- BPMN2. (2011). *Business Process Modeling Notation (BPMN) Version 2.0*. Object Management Group.

- Brezillon P., Mostefaoui G. (2004*). Context-based security policies: a new modeling approach*. In Proceedings of the second IEEE Annual Conference on Pervasive Computing and Communications, p. 154-158. Orlando, FL, USA.

- Briol P. (2008). *Ingénierie des processus métiers, De l'élaboration à l'exploitation*. Lulu.com.

- Brucker A. D., Doser J. (2007). *Metamodel-based UML Notations for Domain Specific Languages*. In 4th International Workshop on Language Engineering.

- Brucker A. D., Hang I., Lückemeyer G., Ruparel R. (2012). *SecureBPMN: Modeling and Enforcing Access Control Requirements in Business Processes*. In Proceedings of the 17th ACM Symposium on Access Control Models and Technologies, p. 123-126. New York, NY, USA.

- Butler W., Lampson. (1971). *Protection*. In Proc. 5th Princeton Conference on Information Sciences and Systems, p. 437-443, Princeton University.

- B-Core (UK) Ltd. (1996). *B-Toolkit User's Manual*, Release 3.2.

- Camus F. (2014). *Profils UML2 et Types de données*. (http://lgl.isnetne.ch/Sagex35793/profilUML/ProfilsUML.pdf)

- Chen Z., Motet G. (2009). *A Language-Theoretic View on Guidelines and Consistency Rules of UML*. Model Driven Architecture - Foundations and Applications, LNCS, vol. 4530, p. 66–81. Springer Berlin Heidelberg.

- Chiorean D., Paşca M., Cârcu A., Botiza C., Moldovan S. (2004). *Ensuring UML Models Consistency Using the OCL Environment*. Electronic Notes in Theoretical Computer Science, vol. 102, p. 99-110.

- ClearSy. (2014a). *Atelier B : Manuel Utilisateur*, Version 4.0.

- ClearSy. (2014b). *Manuel de Référence du langage B*, Version 1.8.8.

- Davis .R, Brabander E. (2007). *ARIS Design Platform: Getting Started with BPM*. Springer-Verlag London.

- Dijkman R. M., Dumas M., Ouyang C. (2008). *Semantics and analysis of business process models in BPMN*. Information and Software Technology, vol. 50, no 12, p. 1281 - 1294.

- Dubauskaite R., Vasilecas O. (2013). *Method on specifying consistency rules among different aspect models, expressed in UML*. Electronics and Electrical Engineering, vol. 19, no 3.

- Dury A., Boroday S., Petrenko A., Lotz V. (2007). *Formal Verification of Business Workflows and Role Based Access Control Systems*. In Proceedings of the International Conference on Emerging Security Information, Systems, and Technologies, p. 201–210. Washington, DC, USA, IEEE Computer Society.

- Elaasar M., Briand L. (2004). *An overview of UML consistency management*. Rapport technique no SCE-04-18, Department of Systems and Computer Engineering, Canada.

- Engels G., Heckel R., Küster J. M. (2001). *Rule-Based Specification of Behavioral Consistency Based on the UML Meta-model*. The Unified Modeling Language. Modeling languages, concepts, and tools, LNCS, vol. 2185, p. 272–286. Springer Berlin Heidelberg.

- Engels G., Heckel R., Küster J. M., Groenewegen L. (2002a). *Consistency-Preserving Model Evolution through Transformations*. The unified modeling language: Model engineering, concepts, and tools, LNCS, vol. 2460, p. 212–227. Springer Berlin Heidelberg.

- Engels G., Küster J. M., Heckel R., Groenewegen L. (2002b). *Towards Consistency-preserving Model Evolution*. In Proceedings of the International Workshop on Principles of Software Evolution, p. 129–132. New York, NY, USA.

- Eshuis H. (2002). *Semantics and Verification of UML Activity Diagrams for Workflow Modelling*. Thèse de doctorat, Université de Twente, Enschede.

- Eshuis R.,Wieringa R. (2001). *A Real-Time Execution Semantics for UML Activity Diagrams*. Fundamental Approaches to Software Engineering, LNCS, vol. 2029, p. 76–90. Springer Berlin Heidelberg.

- Facon P., Laleau R. (1995). *Des spécifications informelles aux spécifications formelles : compilation ou interprétation ?* In Actes du 13ème congrès Inforsid, p. 47-62. Grenoble, France.

- Farail P., Gaufillet P., Canals A., Le Camus C., Sciamma D., Michel P. (2006). *The TOPCASED project: a Toolkit in Open source for Critical Aeronautic SystEms Design*. In European Congress on Embedded Real-Time Software (ERTS). Toulouse, France.

- Feather M. S., Fickas S., Finkelstein A., Lamsweerde A. V. (1997). *Requirements and Specification Exemplars*. Automated Software Engineering, vol. 4, no 4, p. 419-438.

- Ferraiolo D., Kuhn D., Chandramouli R. (2003). *Role-Based Access Control*. Artech House.

- Fingar P., Bellini J. (2004). *The Real-Time Enterprise*. Meghan-Kiffer Press.

- Fontan B. (2008). *Méthodologie de conception de systèmes temps réel et distribués en contexte UML/SysML*. Thèse de Doctorat. Université Paul Sabatier - Toulouse III, France.

- Futatsugi K., Goguen J. A., Jouannaud J. P., Meseguer J. (1985). *Principles of OBJ2*. Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, New Orleans, Louisiana, USA, p. 52-66.ACM.

- Gaaloul K. (2010). *Une approche sécurisée pour la délégation dynamique de tâches dans les systèmes de gestion de workflow*. Thèse de doctorat. Université Henri Poincaré, France.

- Gaaloul K., Proper E., Charoy F. (2012). *An Extended RBAC Model for Task Delegation in Workflow Systems.* In Workshops on Business Informatics Research, Lecture Notes in Business Information Processing, vol. 106, p. 51–63. Springer Berlin Heidelberg.

- Geambasu C. (2012). *BPMN vs. UML activity diagram for business process modeling*. Accounting and Management Information Systems, vol. 11, no 4, p. 637–651.

- Goedertier S., Vanthienen J. (2006). *Compliant and Flexible Business Processes with Business Rules*. In Proceedings of the CAiSE*06, Workshop on Business Process Modelling, Development, and Support. Luxemburg.

- Gogolla M., Kuhlmann M., Hamann L. (2009). *Consistency, Independence and Consequences in UML and OCL Models*. Tests and Proofs, vol. 5668, p. 90-104. Springer Berlin Heidelberg.

- Goncalves G., Hemery F. (2000). *Des cas d'utilisation en UML à la gestion de rôles dans un système d'information*. In Actes du xviiième congrès Inforsid, p. 367–379. Lyon, France.

- Gryce C., Finkelstein A., Nentwich C. (2002). *Lightweight checking for UML based software development*. In Workshop on Consisteny Problems in UML-Based Software Development. Dresden, German.

- Hazem L., Levy N., Marcano-Kamenoff R. (2004). UML2B : *Un outil pour la génération de modèles formels*. In AFADL'2004 - Session Outils.

- Hoare C.A.R. (1978). *Communicating Sequential Processes*. Communications of the ACM, vol. 21, no. 8, p. 666 - 677.

- Huzar Z., Kuzniarz L., Reggio G., Sourrouille J. L. (2005). *Consistency Problems in UML Based Software Development,* LNCS, vol. 3297, p. 1-12. Springer Berlin Heidelberg.

- Idani A. (2006). *B/UML : Mise en relation de spécifications B et de descriptions UML pour l'aide à la validation externe de développements formels en B.* Thèse de Doctorat. Université Joseph Fourier, France.

- Idani A., Ledru Y. (2015). *B for Modeling Secure Information Systems - the B4MSecure platform*. In The 17th International Conference on Formal Engineering Methods, LNCS, vol. 4907. Springer.

- ISO/DIS19440.2. (2007). *Enterprise Integration Constructs For Enterprise Modelling*. (http://www.iso.org/iso/catalogue_detail.htm?csnumber=33834)

- Jürjens J. (2004). *Secure systems development with UML*. Berlin, Heidelberg, Springer-Verlag.

- Kandala S., Sandhu R. (2002). *Secure Role-Based Workflow Models*. In Database and Application Security XV, vol. 87, p. 45-58. Springer.

- Kherbouche M. O. (2013). *Contribution to the business process evolution management.* Thèse de doctorat. Université du Littoral Côte d'Opale.

- Kim S. K., Carrington D., Duke R. (2001). *A Metamodel-based transformation between UML and Object-Z.* In IEEE 2001 symposia on human centric computing languages and environments (HCC'01), p. 112–122. IEEE Computer Society Press.

- Kuzniarz L., Staron M. (2003). *Inconsistencies in student designs*. In Workshop on Consistency Problems in UML-based software development II. Blekinge Institute of Technology.

- Lakhal F., Dubois H., Rieu D. (2012). *Evolution des profils UML : vers une migration automatisée et une optimisation assistée des modèles*. In Actes du Congrès Inforsid, p. 31–46. Montpellier, France.

- Laleau R. (2002). *Conception et développement formels d'applications bases de données*. HDR. Université d'Evry, France.

- Laleau R., Mammar A. (2000). *An Overview of a Method and Its Support Tool for Generating B Specifications from UML Notations*. In 15th IEEE International Conference on Automated Software Engineering, p. 269-272. IEEE Computer Society Press.

- Laleau R., Polack F. (2008). *Using formal metamodels to check consistency of functional views in information systems specification*. Information and Software Technology, vol. 50, no 7–8, p. 797 - 814.

- Lam V. S. W., Padget J. (2005). *Consistency Checking of Sequence Diagrams and Statechart Diagrams Using the pi-calculus*. In Proceedings of the 5th International Conference on Integrated Formal Methods, p. 347–365. Springer-Verlag.

- Lamsweerde A. van. (2007). *Engineering Requirements for System Reliability and Security*, vol. 9, p. 196-238. IOS Press.

- Lange C., Chaudron M. R. V., Muskens J., Somers L. J., Dortmans H. M. (2003). *An empirical investigation in quantifying inconsistency and incompleteness of UML designs.* In Workshop on consistency problems in UML-based software development. Blekinge Institute of Technology.

- Ledang H. (2002). *Traduction systématique de spécifications UML en B*. Thèse de doctorat. Université de Nancy 2, France.

- Ledang H., Souquières J., Charles S. (2003). *Argo/UML+B: un outil de transformation systématique de spécification UML en B*. In AFADL'2003, p. 3–18. IRISA, Rennes.

- Ledru Y., Idani A., Richier J.-L. (2015). *Validation of a Security Policy by the Test of Its Formal B Specification: A Case Study*. In Proceedings of the third FME Workshop on Formal Methods in Software Engineering, p. 6–12. Piscataway, NJ, USA, IEEE Press.

- Legeard B., Peureux F., Utting M. (2002). *Automated boundary testing from Z and B.* In FME'02 (Formal Methods Europe), vol. 2391. Springer.

- Le Thi T. T. (2011). *Modélisation en UML/OCL des langages de programmation et de leurs propriétés et processus IDM*. Thèse de doctorat. Institut de Recherche en Informatique de Toulouse (IRIT), France.

- Leuschel M., Butler M. (2008). *ProB: An Automated Analysis Toolset for the B Method*. International Journal on Software Tools for Technology Transfer, vol. 10, no 2, p. 185–203.

- Lodderstedt T., Basin D., Doser J. (2002). SecureUML: *a UML-based modeling language for model-driven security*. UML, 5th International Conference, Dresden, Germany, 2002. Springer-Verlag London.

- Lübke D., Schneider K. (2008). *Visualizing Use Case Sets as BPMN Processes*. In IEEE Conference on Requirements Engineering Visualization, p. 21-25. Barcelona, Catalunya.

- Ma G.,Wu K., Zhang T., Li W. (2011). *A flexible policy-based access control model for Workflow Management Systems*. In IEEE International Conference on Computer Science and Automation Engineering (CSAE), vol. 2, p. 533-537.

- Malgouyres H. (2007). *Définition et détection automatique des incohérences structurelles et comportementales des modèles UML - Couplage des techniques de métamodélisation et de vérification basée sur la programmation logique-*. Thèse de doctorat. INSA de Toulouse, France.

- Mammar A. (2002). *Un environnement formel pour le développement d'applications bases de données*. Thèse de doctorat. CNAM-Paris, France.

- Mammar A., Laleau R. (2004). *UML2SQL: Un environnement intégré pour le développement d'implémentations relationnelles à partir de diagrammes UML*. In AFADL'2004 - Session outils. Besançon, France.

- Marcano R. (2002). *Spécification formelle d'objets en UML/OCL et B : une approche transformationnelle*. Thèse de doctorat. Université de Versailles, France.

- Mar Gallardo M. D., Merino P., Pimentel E. (2002). *Debugging UML Designs with Model Checking*. Journal of Object Technology, vol. 1, p. 101–117.

- Matulevicius R., Dumas M. (2011). *Towards Model Transformation between SecureUML and UMLsec for Role-based Access Control*, vol. 224, IOS Press Ebooks.

- MEGA. (2004). *Business process modeling and standardization*. (http://216.197.108.171/publicationfiles/1204%20WP%20BPM%20and%20Standardization%20-%20Lonjon.pdf)

- MEGA. (2008). *Modélisation des processus métiers et standardisation*. (http://www.bpms.info/modelisation-des-processus-metiers-et-standardisation/)

- Meyer E. (2001). *Développements formels par objets : utilisation conjointe de B et d'UML*. Thèse de doctorat. Université de Nancy 2, France.

- Milhau J. (2011). *Un processus formel d'intégration de politiques de contrôle d'accès dans les systèmes d'information*. Thèse de doctorat. Université Paris-Est, France.

- Milhau J., Idani A., Laleau R., Labiadh M., Ledru Y., Frappier M. (2011). *Combining UML, ASTD and B for the formal specification of an access control filter*. Innovations in Systems and Software Engineering, vol. 7, no 4, p. 303–313. Springer.

- Miller J., Mukerji J. (2003). *MDA guide*. Object Management Group.

- Miller T., Strooper P. (2001). *Animation can show only the presence of errors, never their absence*. In Software Engineering Conference, p. 76-85. Canberra, ACT.

- Mingsong C., Xiaokang Q., Xuandong L. (2006). *Automatic Test Case Generation for UML Activity Diagrams*. In Proceedings of the International Workshop on Automation of Software Test, p. 2–8. New York, NY, USA.

- Montrieux L., Wermelinger M., Yu Y. (2011). *Tool Support for UML-Based Specification and Verification of Role-Based Access Control Properties*. In 8th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, p. 456-459. ACM New York, NY, USA.

- Morley C., Hugues J., Leblanc B., Hugues. O.(2005). *Processus métiers et S.I. Evaluation, modélisation, mise en œuvre*. DUNOD.

- Mouratidis H., Giorgini P. (2007). *Secure Tropos: A Security-Oriented Extension of the Tropos Methodology*. International Journal of Software Engineering and Knowledge Engineering, vol. 17, no 2, p. 285–309.

- Métayer C. (2010). *AnimB Homepage*. (http://www.animb.org/index.xml)

- Muehlen M. Z. (2004). *Workflow-based Process Controlling: Foundation, Design, and Implementation of Workflow-Driven Process Information Systems*. Logos Verlag.

- Nguyen H. P. (1998). *Dérivation de spécifications formelles B à partir de spécifications semiformelles*. Thèse de doctorat. CNAM-Paris, France.

- Nurcan S., Grosz G., Souveyet C. (1998). *Describing Business Processes with a Guided Use Case Approach*. Advanced Information Systems Engineering, LNCS, vol. 1413, p. 339-362. Springer Berlin Heidelberg.

- OCL2. (2012). *Object Constraint Language (OCL) Version 2.3.1*. Object Management Group. (http://www.omg.org/spec/OCL/2.3.1/PDF/)

- Oh S., Park S. (2003). *Task-role-based Access Control Model*. Information System, vol. 28, no 6, p. 533–562.

- Ouhammou Y., Keulkeul M. B. alias. (2012). *Apprendre à créer un profil UML avec Eclipse Papyrus*. (http://yassineouhammou.developpez.com/tutoriels/eclipse/uml-profil-papyrus/).

- Owre S., Rushby J., Shankar N. (1992). *PVS: A Prototype Verification System*. In 11th International Conference on Automated Deduction (CADE). Lecture Notes in Artificial Intelligence, vol. 607, p. 748–752. Springer-Verlag, Saratoga, NY.

- Pesic M., Aalst W. M. P. (2006). *A Declarative Approach for Flexible Business Processes Management*. Business process management workshops, LNCS, vol. 4103, p. 169-180. Springer Berlin Heidelberg.

- Petriu D. C., Sun Y. (2000). *Consistent Behaviour Representation in Activity and Sequence Diagrams*. The Unified Modeling Language : Advancing the Standard, LNCS, vol. 1939, p. 369–382. Springer Berlin Heidelberg.

- Potet M-L.(2002). *Spécifications et développements formels: Etude des aspects compositionnels dans la méthode B*. HDR, Institut National Polytechnique de Grenoble.

- Rasch H., Wehrheim H. (2003). *Checking Consistency in UML Diagrams: Classes and State Machines*. Formal Methods for Open Object-Based Distributed Systems, LNCS, vol. 2884, p. 229–243. Springer Berlin Heidelberg.

- Rodríguez A., Fernández-Medina E., Piattini M. (2007). *A BPMN Extension for the Modeling of Security Requirements in Business Processes*. IEICE - Transactions on Information and Systems, vol. E90-D, no 4, p. 745-752. ACM.

- Rodríguez A., Fernández-Medina E., Trujillo J., Piattini M. (2011). *Secure business process model specification through a UML 2.0 activity diagram profile*. Decision Support Systems, vol. 51, no 3, p. 446 - 465.

- Roques P. (2006). *UML 2 par la Pratique*. Eyrolles, Paris.

- Rosemann M., Recker J., Flender C. (2008). *Contextualisation of business processes*. International Journal of Business Process Integration and Management, vol. 3, no 1, p. 47-60.

- Russell N., Aalst W. van der, Hofstede A. ter, Wohed P. (2006). *On the suitability of UML 2.0 activity diagrams for business process modelling*. In 3rd Asia-Pacific Conference on Conceptual Modelling, vol.53, p. 95-104, ACM.

- Sarshar K., Loos V. (2007). *Modeling the Resource Perspective of Business Processes by UML Activity Diagram and Object Petri Net*. Enterprise Modeling and Computing with UML, (http://www.igi-global.com/chapter/modeling-resource-perspective-business-processes/18409).

- Sbaï Z. (2010). *Contribution à la Modélisation et à la Vérification de Processus Workflow*. Thèse de doctorat. CNAM de Paris, France.

- Schefer S., Strembeck M. (2011). *Modeling Support for Delegating Roles, Tasks, and Duties in a Process-Related RBAC Context.* In International Workshops on Advanced Information Systems Engineering (CAiSE), London, UK, vol. 83, p.660–667. Springer Berlin Heidelberg.

- Schefer-Wenzl S., Strembeck M. (2012). *Modeling Context-Aware RBAC Models for Business Processes in Ubiquitous Computing Environments*. In Third FTRA International Conference on Mobile, Ubiquitous, and Intelligent Computing (MUSIC), p. 126-131.

- Servat T. (2006). *BRAMA: A New Graphic Animation Tool for B Models*. B 2007: Formal Specification and Development in B, LNCS, vol. 4355, p. 274–276. Springer Berlin Heidelberg.

- Silva P. P. D. (2001). *A Proposal for a Lotos-Based Semantics for UML*. Rapport technique no UMCS-01-06-1. Dept. of Computer Science, Univ. Manchester.

- Sindre G. (2007). *Mal-Activity Diagrams for Capturing Attacks on Business Processes.* Requirements Engineering: Foundation for Software Quality, LNCS, vol. 4542, p. 355–366. Springer Berlin Heidelberg.

- Snook C., Butler M. (2004). *U2B-A tool for translating UML-B models into B*. UML-B specification for proven embedded systems design, p. 85-108. Springer.

- Snook C., Butler M. (2006). *UML-B: Formal modeling and design aided by UML*. ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 15, no 1, p. 92–122.

- Souag A., Salinesi C., Mazo R., Comyn-Wattiau I. (2015). *A security ontology for security requirements elicitation*. In 7th International Symposium on Engineering Secure Software and Systems, p. 157–177. Milan, Italy.

- Sourrouille J. L., Caplat G. (2002). *A pragmatic view on consistency checking of UML models.* In Proc. workshop on consistency problems in UML-based software development. Dresden, German.

- Sourrouille J. L., Caplat G. (2003). *A Pragmatic View on Consistency Checking of UML Models*. In Proc. workshop on consistency problems in UML-based software development. Blekinge Institute of Technology.

- Spivey J. M. (1989). *The Z Notation: A Reference Manual*. Prentice-Hall, Upper Saddle River, NJ, USA.

- Straeten R., Jonckers V., Mens T. (2006). *A formal approach to model refactoring and model refinement*. Software & Systems Modeling, vol. 6, no 2, p. 139–162.

- Strembeck M., Mendling J. (2011). *Modeling process-related RBAC models with extended UML activity models*. Information and Software Technology, vol. 53, no 5, p. 456-483.

- Störrle H., Hausmann J. H., Paderborn U. (2005). *Towards a formal semantics of UML 2.0 activities*. In Proceedings German Software Engineering Conference, p. 117–128.

- Thomas R. K, Sandhu R.(1993). *Towards a task-based paradigm for flexible and adaptable access control in distributed applications*. 2nd New Security Paradigms Workshop, p 138-142. ACM New York.

- Toro R. A. L. (2009). *Estimation des risques d'incohérence liés à l'emploi d'UML pour le développement des système*s. Thèse de doctorat. INSA de Toulouse, France.

- Torre D., Labiche Y., Genero M. (2014). *UML consistency rules: a systematic mapping study*. In 18th International Conference on Evaluation and Assessment in Software Engineering, p. 1-10. New York, NY, USA.

- Tripathy A., Mitra A. (2012). *Test Case Generation Using Activity Diagram and Sequence Diagram*. In Proceedings of International Conference on Advances in Computing, vol. 174, p. 121–129. New Delhi, Springer.

- Truong N. T. (2006). *Utilisation de B pour la vérification de spécifications UML et le développement formel orienté objet*. Thèse de doctorat. Université Nancy II, France.

- Ulmer J. S. (2011). *Approche générique pour la modélisation et l'implémentation des processus*. Thèse de Doctorat. INP de Toulouse, France.

- UML2. (2011). *Unified Modeling Language: Superstructure (version 2.4)*. Object Management Group.

- Vidal J.-P. S. (2006). *Maîtrise de la cohérence des modèles UML d'applications critiques. Approche par l'analyse des risques liés au langage UML*. Thèse de Doctorat. Institut National des Sciences Appliquées de Toulouse, France.

- Wainer J., Barthelmess P., Kumar A. (2003). *W-RBAC. A workflow security model incorporating controlled overriding of constraints*. International Journal of Cooperative Information Systems, vol. 12, no 4, p. 455-486.

- Wainer J., Kumar A., Barthelmess P. (2007). *DW-RBAC: A formal security model of delegation and revocation in workflow systems*. Information Systems, vol. 32, no 3, p. 365-384.

- WFMC. (1999). *Workflow management coalition Terminology and glossary. Workflow Management Coalition*. (http://www.wfmc.org/standards/docs/TC-1011_term_glossary_v3.pdf)

- WFMC. (2001). *Workflow Security Considerations*.Workflow Management Coalition. (WFMCTC- 1019)

- Wolter C., Miseldine P., Meinel C. (2009). *Verification of Business Process Entailment Constraints Using SPIN*. Engineering Secure Software and Systems, LNCS, vol. 5429, p. 1-15. Springer Berlin Heidelberg.

- Wolter C., Schaad A. (2007). *Modeling of Task-Based Authorization Constraints in BPMN*. Business Process Management, LNCS, vol. 4714, p. 64-79. Springer Berlin Heidelberg.

- Wong P. Y., Gibbons J. (2011). *Formalisations and applications of BPMN*. Science of Computer Programming, vol. 76, no 8, p. 633 - 650.

- Yuan E., Tong J. (2005). *Attributed Based Access Control (ABAC) for Web Services*. In Proceedings of the IEEE International Conference on Web Services, p. 561-569.

- Zefouni S. (2012). *Aide à la conception de workflows personnalisés: Application à la prise en charge à domicile*. Thèse de doctorat. Université Toulouse III Paul Sabatier, France.

- Zeng L., Flaxer D., Chang H., Jeng J.J. (2002). *PLM_{flow}—Dynamic Business Process Composition and Execution by Rule Inference*. Technologies for E-Services, LNCS, vol. 2444, p. 141–150. Springer Berlin Heidelberg.

Publications issues de cette Thèse

1. Chehida S. (2016). *Approche de spécification et validation formelles de politiques RBAC au niveau des processus métiers*, In AFADL 2016, Besançon, France.
2. Chehida S., Idani A., Ledru Y., Rahmouni M.K. (2016). *Combining UML and B for the specification and validation of RBAC policies in Business Process activities*, In IEEE RCIS 2016, Grenoble, France. http://ieeexplore.ieee.org/document/7549284/?arnumber=7549284.
3. Chehida S., Idani A., Ledru Y., Rahmouni M.K. (2016). *Extensions du diagramme d'activité pour la spécification de politiques RBAC* , Ingénierie des Systèmes d'Information, volume 21, n°2/2016, page 11-37. http://isi.revuesonline.com/article.jsp?articleId=36419.
4. Chehida S., Idani A., Ledru Y., Rahmouni M.K. (2015). *Extensions du diagramme d'activité pour contrôler l'accès au SI* . In INFORSID15, p. 151-165. Biarritz, France. http://inforsid.fr/Biarritz2015/wp-content/uploads/actes2015/mm2-2.pdf.
5. Chehida S., Rahmouni M.K. (2013). *An UML-based Approach for the Security of Information Systems*, In ICIST-2013, Université Abdelmalek Essaadi, Tangier, Morocco.
6. Chehida S., Rahmouni M.K. (2012). *UML Extensions for Security Requirements Analysis of Web Applications*, In IACeT'2012, Zarqa University – Jordan.
7. Chehida S., Rahmouni M.K. (2012). *Security Requirements Analysis of Web Applications using UML*, In ICWIT'2012, Université de Sidi Bel Abbes, Algerie. http://ceur-ws.org/Vol-867/Paper24.pdf.
8. Chehida S., Rahmouni M.K. (2012). *UML pour la sécurité à priori des Applications Web »*, In CITIM2012, Université de Mascara, Algérie.
9. Chehida S., Rahmouni M.K. (2011). *Towards an Approach for the Security of Information Systems with UML*, In ACIT2011, Naif Arab University for Security Science (NAUSS), Riyadh, Saudi Arabia. http://www.nauss.edu.sa/acit/PDFs/f2816.pdf.
10. Chehida S., Rahmouni M.K. (2010). *Vers la spécification des exigences de sécurité des systèmes d'information*, In COSI 2010, Université Kasdi Merbah de Ouargla, Algérie.

Résumé

La spécification de politiques de contrôle d'accès constitue une étape cruciale dans le développement d'un SI. Le présent travail propose une approche qui combine les langages UML et B pour la spécification et la validation de politiques RBAC au niveau des activités d'un processus métier. Notre approche commence par la spécification semi-formelle des règles de contrôle d'accès à l'aide de notre extension de diagrammes d'activités d'UML2 nommée BAAC@UML (Business Activity Access Control with UML). Les diagrammes d'activité sont définis à deux niveaux : un niveau abstrait qui ne détaille pas les règles d'autorisation et un niveau concret où des contraintes sont associées à certaines actions ou à l'ensemble du diagramme. Notre profil introduit les concepts de *rôle* et *pré-condition contextuelle de contrôle d'accès* pour garder l'accès des utilisateurs aux activités. Nous avons défini un méta-modèle dans le but de spécifier la sémantique de nos diagrammes d'activité ainsi que la sémantique de leurs liens avec les modèles Secure UML qui permettent la spécification de la vue statique de politique RBAC. Nous avons également proposé un ensemble de règles, exprimées en OCL, qui permet d'assurer la cohérence structurelle entre les modèles d'activité concrète et les modèles SecureUML. Les modèles BAAC@UML sont ensuite traduits en spécifications formelles exprimées en langage B en utilisant la plateforme B4MSecure. La spécification B définit un cadre rigoureux favorisant la vérification et la validation (V&V) de la politique de sécurité. Nous utilisons l'animateur ProB et le prouveur AtelierB dans les activités de V&V. La preuve assure la conservation des contraintes de sécurité invariantes dans l'exécution de l'activité. L'animation permet de simuler l'exécution des scénarios d'activité par les utilisateurs.

MOTS-CLÉS :
RBAC ; Processus métier; SecureUML; Diagramme d'activités d'UML2; Cohérence; OCL; Formalisation; Langage B; Animation; Preuve.